Eduardo A. Barbosa

INTRODUÇÃO À

Linguagem de Máquina para

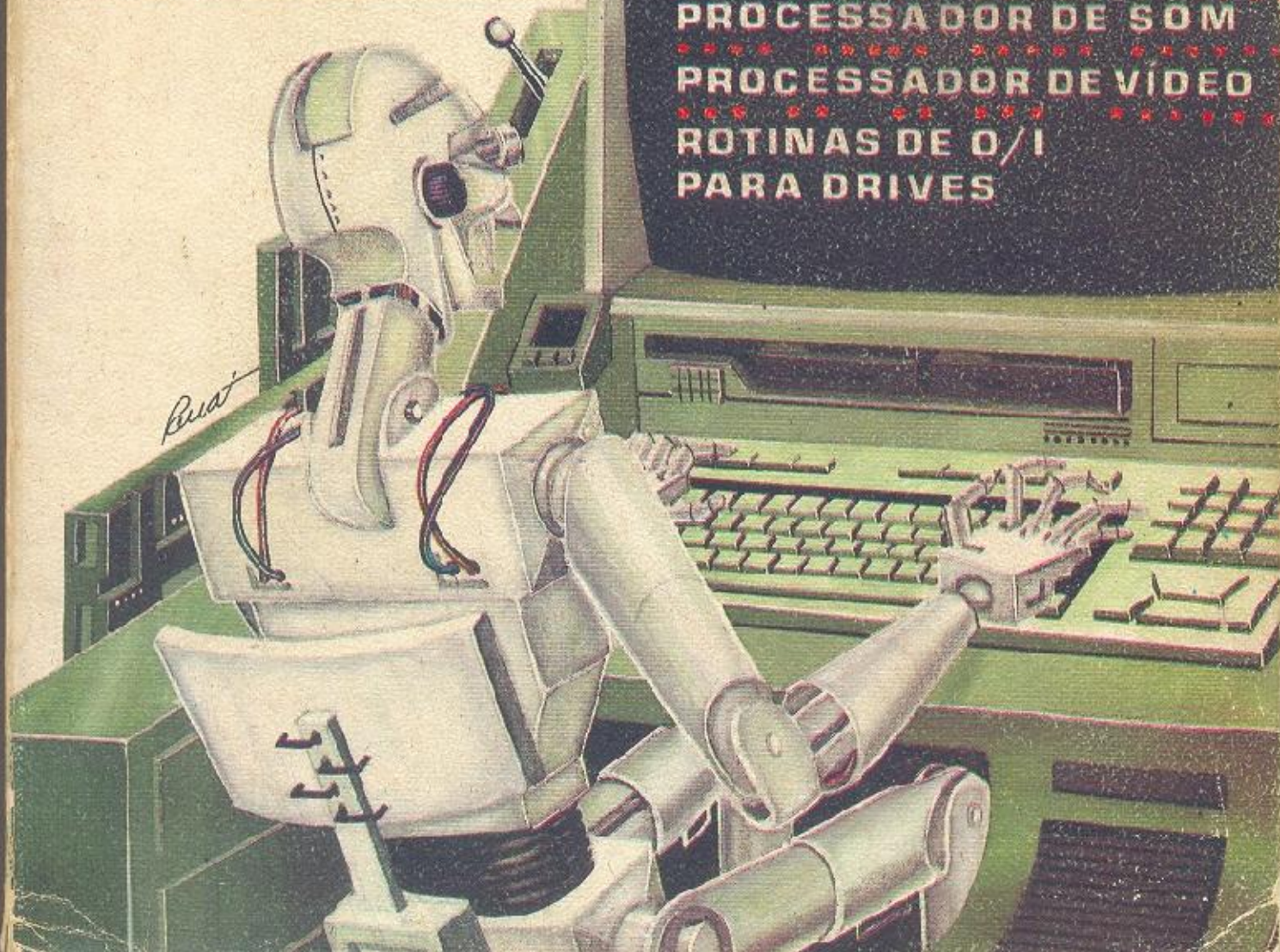
ASSEMBLY Z80

INTERAÇÃO ENTRE AS LINGUAGENS BASIC E ASSEMBLY

INTERFACE PROGRAMÁVEL DE PERIFÉRICOS-PPI

PROCESSADOR DE SOM

PROCESSADOR DE VÍDEO

ROTINAS DE O/I PARA DRIVES

CIÊNCIA MODERNA COMPUTAÇÃO LTDA.

INTRODUÇÃO À LINGUAGEM DE MÁQUINA PARA MSX

EDUARDO ALBERTO BARBOSA

DEDICATÓRIA

Dedico este livro aos meus pais e a minha irmã, que sempre procuraram me mostrar que apesar de tudo, a vida deve ser encarada com otimismo, sendo esta a única forma de realmente construirmos algo. A vocês, minha família, a minha eterna grat dão pelo amor e carinho que sempre me dispensaram.

AGRADECIMENTO

Gostaria de agradecer a todos os amigos que me incentivaram a escrever este livro, coloborando com valiosas sugestões.

Especial agradecimento ao meu amigo Paulo André Pitanga Marques, o meu maior incentivador, sem o qual esta obra não teria sido realizada. A você, meu amigo, o meu mais profundo respeito e admiração e os meus maiores agradecimentos.

Eduardo Alberto Barbosa

PREFÁCIO

O objetivo deste livro é o de principalmente, desvendar o mundo da linguagem de máquina, quase sempre encoberto por uma nuvem de mistério. Programar em linguagem de máquina não é difícil, apenas exige um pouco mais de atenção, pois por não ser uma linguagem interpretada como o BASIC, não possui mensagens de erro. Desta forma um pequeno erro pode causar situações imprevisíveis, desde a execução errada até um reset do sistema. Outro detalhe que merece ser destacado, é que programar em linguagem de máquina não estraga o computador, a única exceção está comentada no capítulo sobre a interface programável de periféricos.

Este livro foi dividido em capítulos de ordem crescente de complexidade, assim sendo, só passe para o capítulo posterior, se tiver certeza de que entendeu tudo muito bem. Nos últimos capítulos aproveitamos todo o conhecimento adquirido até então, para fazer pequenos exemplos de como programar em linguagem de máquina. Execute estes exemplos e tente fazer modificações nos programas apresentados, só assim

você irá ganhar a prática em programação assembly.

Para finalizar, eu me coloco a sua inteira disposição para tirar qualquer dúvida que porventura tiver. Para tanto, escreva para a **CIÊNCIA MODERNA COMPUTAÇÃO LTDA**, discrevendo a sua dúvida.

SUMÁRIO

CAPÍTULO 1

CONCEITOS BÁSICOS

INTRODUÇÃO

Este livro foi concebido para introduzir qualquer programador na linguagem BASIC do MSX, a linguagem do seu computador, ou seja, a linguagem de máquina.

Como você já deve saber, o cérebro do seu computador é um chip conhecido como **Z80**. A aparência deste chip é a de uma caixa preta com 40 pernas que fazem o contato do Z80 com os demais componentes do microcomputador. A troca de informações entre o Z80 e os demais componentes é feita por intermédio de 8 pinos que constituem o barramento de dados (D0, ..., D7). Esta troca de informações se realiza de acordo com as combinações elétricas que recebem estes 8 pinos.

Já que estamos falando de combinações elétricas, vamos introduzir o conceito de sinais elétricos. A presença de um sinal é representada

pelo número "1", o que quer dizer que o ponto analisado apresenta uma tensão de 5 volts (variável de acordo com a tecnologia empregada), já a ausência de sinal é representada pelo número "0", o que indica uma tensão de 0 volts.

Como sabemos, o Z80 possui 8 pinos que podem receber 256 (2 ^ 8) combinações de "0" e "1". Agora você já está apto a perceber o motivo do Z80 pertencer à classe dos microprocessadores de 8 bits, onde o termo bit significa dígito binário, que, como vimos, pode assumir o valor 0 ou 1.

A cada grupo de 8 bits dá-se o nome de byte. Para entender estes conceitos preste atenção no esquema abaixo.

bits 10001111
76543210

Como você poderá observar, o bit 7 apresenta o valor 1 bem como os bits 3,2,1 e 0. Já os bits 6,5 e 4 apresenta o valor 0. Como dissemos cada grupo de 8 bits representa um byte, vamos então ver o valor do byte representado pela seqüência de bits acima.

BYTE = 1 * 2 ^ 7 + 0 * 2 ^ 6 + 0 * 2 ^ 5 + 0 * 2 ^ 4 + 1 * 2 ^ 3 +
1 * 2 ^ 2 + 1 * 2 ^ 1 + 1 * 2 ^ 0 = 143

Após esta breve introdução passaremos ao desenvolvimento dos conceitos básicos necessários a todos os jovens programadores da linguagem de máquina.

VANTAGENS E DESVANTAGENS DA LINGUAGEM DE MÁQUINA

A esta altura você deve estar se perguntando se foi realmente necessário aprender os conceitos de bit e byte. Acontece que as diferentes

combinações que assumem os 8 bits antes mencionados são as que indicam ao Z80 qual instrução deverá ser realizada. Você deve estar lembrado de que no tópico anterior mencionei que são possíveis 256 combinações diferentes, o que poderia nos conduzir ao raciocínio de que seriam possíveis 256 instruções no Z80. Acontece porém, que a realidade felizmente é outra. Ao planejar o Z80, a ZILOG (fabricante do Z80), reservou quatro instruções das 256 possíveis para indicar que o próximo byte deve ser interpretado como instrução ao invés de endereço ou dados. Assim sendo, o número de instruções oficiais é de 694. Digo oficiais porque existem mais 438 não divulgadas pela ZILOG e que por isso, não são reconhecidas pelos montadores assembler comerciais.

Diante de tantas instruções você deve estar se perguntando se não é difícil programar em linguagem de máquina. Eu lhe asseguro que não, embora o início da aprendizagem seja um pouco penoso. Quanto às vantagens de se programar em linguagem de máquina, irei enumerar as três que julgo as mais importantes, que são:

1. Os programas em linguagem de máquina são sensivelmente mais rápidos.

2. Os programas em linguagem de máquina em geral ocupam menos memória.

3. Não há necessidade de utilizarmos o interpretador BASIC, que é muito lento.

Mas, como afirmei, o início é um pouco penoso. Vamos ver os motivos:

1. Por não passarem por nenhum interpretador torna-se difícil detectar possíveis erros que os programas possam apresentar.

2. Como dependem muito da máquina na qual são escritos, torna-se muito difícil, senão impossível, usá-los em outros computadores.

3. Para executarmos comandos simples, como os comandos SAVE e LOAD, necessitamos de um número razoável de instruções.

4. Os programas que usam aritmética de ponto flutuante são bastante difíceis de serem programados.

Após esta breve análise sobre as vantagens e desvantagens de se programar em código de máquina, passaremos à etapa de como instruir o Z80.

OS PROGRAMAS MONTADORES

Os montadores assembler nada mais são do que programas que nos auxiliam na programação em linguagem de máquina. Não fossem eles e nós teríamos que programar o Z80 introduzindo na memória do computador seqüências de zeros e uns que embora signifiquem instruções para o Z80, para nós nada representam. Diante deste fato, os programadores da própria ZILOG conceberam uma maneira de representar essas seqüências de zeros e uns, com termos curtos (em inglês), que correspondessem às instruções realizadas. Cada instrução de código de máquina assim representada recebeu o nome de **mnemônico.** Já temos, portanto, três formas de se representar uma mesma instrução, que são:

1. Instruções na base dois, em bits, por exemplo:

   01110110

2. Instruções na base decimal, por exemplo:

   118

3. Instruções num montador, em mnemônicos por exemplo:

HALT

A instrução utilizada neste exemplo nada mais faz do que instruir o Z80 a permanecer parado até ordem em contrário.

COMO TRABALHA A CPU

Até agora já sabemos que o Z80 trabalha mediante as combinações possíveis assumidas pelos 8 pinos do barramento de dados. Sabemos também que cada um desses pinos só assume dois valores possíveis que são, zero ou um. É por este motivo que se utiliza a base binária nos computadores. A escolha da base binária é bem óbvia pois, como vimos, os sistemas digitais só contam com dois tipos de informações; já nós utilizamos a base decimal por termos 10 dedos. Da mesma forma que nós usamos os nossos dedos para contar, e papel e lápis quando queremos recordar resultados anteriores, o computador usa os seus dedos (registros) para contar e a sua memória para aquivar dados. Continuando com esta nossa comparação, poderíamos afirmar que o Z80 possui 8 mãos com 8 dedos cada e outras 2 mãos com 16 dedos cada. As mãos com 8 dedos seriam os registros A, B, C, D, E, F, H e L, que são conhecidos como registros de 8 bits. Já as 2 mãos de 16 bits são conhecidas como registros de 16 bits que são designados por IX e IY.

Apesar da CPU possuir 10 registros básicos, há ainda uma outra forma de armazenamento utilizada por ela, que é a pilha (stack em inglês). O armazenamento de dados na pilha obedece à regra LIFO (last in first out) que em português poderia ser traduzida para "o primeiro a entrar é o último a sair". A utilização da pilha deve estar envolvida por diversos cuidados pois é o conteúdo da pilha que é usado pelo Z80 no retorno de subrotinas. Por enquanto é bastante que você saiba que a pilha

também pode ser usada para armazenar informações, quanto aos cuidados a tomar, ficarão mais claros nos próximos capítulos.

O HARDWARE DO MSX

Antes de mais nada devemos definir, da maneira mais prática possível, o que é HARDWARE e o que é SOFTWARE.

O termo HARDWARE engloba todos os componentes eletrônicos, ou seja, os componentes físicos que compõem um computador, já o termo SOFTWARE é o termo que designa os programas utilizados por nós usuários, e que são a única maneira do computador nos entender.

O HARDWARE do MSX é composto basicamente por 5 componentes, descritos abaixo:

1. CPU modelo Z80-A
2. Memória (ROM e RAM)
3. Processador de Vídeo (VDP)
4. Gerador de sons (PSG)
5. Interface programável de periféricos (PPI)

Nos próximos tópicos passaremos a descrever cada um destes componentes na mesma ordem apresentada acima.

A UNIDADE DE PROCESSAMENTO CENTRAL (CPU)

Até agora já sabemos que a CPU possui 8 registros de 8 bits, e 2 registros de 16 bits. Dos 8 registros de 8 bits, somente 7 podem ser usados para armazenar dados, pois o registro F é usado pelo sistema para indicar resultados das operações anteriores (zero, vai um, etc.). Se qui-

sermos, podemos unir os registros B e C, D e E, H e L para formar novos registros de 16 bits., que passarão a se chamar BC, DE e HL respectivamente. Dentro da CPU existem mais 8 registros que podemos usar através de um pequeno número de instruções.

Passaremos agora aos demais componentes da CPU. São eles:

SP (Stack Pointer)

A função deste registro é assinalar a posição na memória do último elemento armazenado na pilha. O seu valor é permanentemente atualizado conforme a CPU introduz ou retira dados da pilha.

PC (Prog. Counter)

A função deste registro é a de informar à CPU a posição da próxima instrução em linguagem de máquina na memória. É, por assim dizer, um contador de programa.

ULA

A função da unidade lógica-aritmética pode ser comparada à de uma pequena calculadora de bolso usada pela CPU. A ULA está capacitada para realizar somas e subtrações; desconhece portanto multiplicações e divisões. Além disso é capaz de realizar comparações entre números ou entre registros de 8 bits. Os resultados da ULA são colocados sob a forma de flags (sinalizadores) no registro F. Por mais inteligente que possa parecer, a ULA é inútil sem a ajuda dos demais chips componentes do MSX.

MEMÓRIA

A CPU pode cumprir o seu papel graças ao contador de programa (PC), que por ser um registro de 16 bits, pode endereçar 2 ^ 16 (65536) posições de memória. No MSX existem dois tipos de memória, que são: a ROM (memória somente para leitura) e a RAM (memória de

acesso aleatório). Na memória ROM estão gravadas as rotinas de entrada e saída do MSX, tais como, as rotinas de gravação e leitura de um byte na fita K7, além do interpretador BASIC. A memória ROM apresenta a particularidade de não perder os dados nela gravados quando se corta a alimentação do computador. Outra característica é a de não permitir que o programador altere o seu conteúdo, sendo por esse motivo conhecida como memória somente de leitura. A memória do tipo RAM possui características inversas às da memória ROM, pois perde o conteúdo ao se desligar o computador e permite que o seu conteúdo seja alterado pelo programador. Tanto a RAM como a ROM permitem a gravação de dados de 8 bits.

O PROCESSADOR DE VÍDEO (VDP)

Este componente, também conhecido como TMS-9928, é o responsável pela geração das imagens mediante o controle da CPU. Este chip armazena as informações necessárias para a produção de imagens, numa memória especial do tipo RAM, conhecida como VRAM. Num capítulo posterior analisaremos o funcionamento completo do VDP.

O GERADOR DE SOM (PSG)

O gerador de som é um chip conhecido como AY-3-8910 que, como o próprio nome indica, é o responsável pela geração de sons mediante as ordens da CPU.

A INTERFACE PROGRAMÁVEL DE PERIFÉRICOS (PPI)

A função do PPI no computador MSX é mais complexa que a dos dois chips apresentados anteriormente. Este chip ajuda o Z80 no controle

do teclado, na seleção da memória, no controle do motor do gravador K7, na gravação de bytes na fita K7, no controle da lâmpada de CAPS LOCK e do click do teclado quando pressionamos alguma tecla. Como os dois chips anteriores, este será estudado mais a fundo num capítulo posterior.

CAPÍTULO 2

A ARITMÉTICA DOS COMPUTADORES

Como já vimos, o Z80 é capaz de realizar operações artiméticas simples manipulando dados de 8 bits. Acontece porém, que como você já deve ter desconfiado, fazer uma simples soma de dois números manipulando 8 bits é algo difícil para nós usuários, acostumados a realizar as operações matemáticas mais simples na base decimal. Por esta razão, entre outras, é que se usa a base hexadecimal com mais freqüência que a notação binária, e, até mesmo, que a notação decimal.

A ARITMÉTICA HEXADECIMAL

Como qualquer outro sistema decimal, o sistema hexadecimal possui uma base para representar os diversos valores numéricos. A base deste sistema, como o próprio nome indica, é dezesseis, da mesma forma que no sistema decimal a base empregada é dez, e no sistema binário a base é dois.

A maneira de representar algarismos na base hexadecimal é análoga à da base decimal, só que agora fazemos uso de 16 algarismos. A tabela abaixo vai ajudá-lo a compreender melhor os conceitos vistos até agora.

DECIMAL	BINÁRIO	HEXADECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Baseando-se na tabela anterior podemos tirar algumas conclusões úteis:

1. Cada cadeia de quatro bits pode ser representada por um único algarismo hexadecimal.

2. Para representarmos seqüências de 8 bits necessitaremos de 2 algarismos hexadecimais, e, para representar cadeias de 16 bits, necessitaremos de 4 algarismos hexadecimais.

Vamos agora ver como se transformam números da base hexadecimal para a base decimal e vice-versa.

Seja por exemplo o número 1987 decomposto na base decimal:

```
1987
    7 * 10 ^ 0 =      7
   8 * 10 ^ 1  =     80
  9 * 10 ^ 2   =    900
 1 * 10 ^ 3    =  +1000
                  -----
                   1987
```

Observe agora o mesmo número decomposto na base hexadecimal:

```
1987
    7 * 16 ^ 0 =      7
   8 * 16 ^ 1  =    128
  9 * 16 ^ 2   =   2304
 1 * 16 ^ 3    =  +4096
                  -----
                   6535
```

Logo 1987 na base hexadecimal equivale a 6535 na base decimal. Com este exemplo aprendemos a transformar um número da base hexadecimal para a decimal. A conversão da base decimal para a base hexadecimal é um pouco mais complicada como veremos a seguir:

```
1º Passo: 6535 |16
           135  408
             7          Resto= 7

2º Passo:  408 |16
            88  25
             8          Resto= 8

3º Passo:   25 |16
             9  1       Resto= 9

4º Passo:    1 |16
             1  0       Resto= 1
```

Aglutinando os restos debaixo para cima obtemos o número correspondente na base hexadecimal que é 1987.

A ARITMÉTICA BINÁRIA

As regras para conversão da base binária para a decimal e vice-versa são as mesmas aplicadas na conversão da base hexadecimal para decimal e vice-versa. Assim sendo, veja o número decimal do exemplo anterior:

```
1º  Passo: 6535 |_2____
             05   3267
              13
               15
                1          Resto = 1

2º  Passo: 3267 |_2____
             12   1633
              06
               07
                1          Resto = 1

3º  Passo: 1633 |_2____
              0   816
               3
               13
                1          Resto = 1

4º  Passo:  816 |_2____
            016   408
              0            Resto = 0
```

```
5º  Passo:  408 |2
            008  204
              0         Resto= 0

6º  Passo:  204 |2
            004  102
              0         Resto= 0

7º  Passo:  102 |2
             02  51
              0         Resto= 0

8º  Passo:   51 |2
             11  25
              1         Resto= 1

9º  Passo:   25 |2
             05  12
              1         Resto= 1

10º Passo:   12 |2
              0  6      Resto= 0

11º Passo:    6 |2
              0  3      Resto= 0

12º Passo:    3 |2
              1  1      Resto= 1

13º Passo:    1 |2
              1  0      Resto= 1
```

Aglutinando-se os restos debaixo para cima chegamos ao final da conversão, onde o número 1100110000111B é igual a 6535D, como já podíamos prever. Os sufixos B, D, e H após os algarismos indicam a ba-

se que devemos considerar, respectivamente binária, decimal e hexadecimal.

A ADIÇÃO BINÁRIA

A adição binária obedece as mesmas regras da adição decimal. A figura a seguir ilustra as quatro possíveis ocorrências:

0 + 0= 0
0 + 1= 1
1 + 0= 1
1 + 1= 0 e vai " 1 "

Para ficar mais claro, veja o exemplo:

```
 110100= 52D
+000111=  7D
-----------
 111011= 59D
```

A SUBTRAÇÃO BINÁRIA

As regras da subtração binária também são as mesmas que as da subtração decimal. Observe o exemplo seguinte:

0 - 0= 0
1 - 0= 1
1 - 1= 0
0 - 1= 1 e vai " -1 "

Para que você possa entender melhor acompanhe o exemplo:

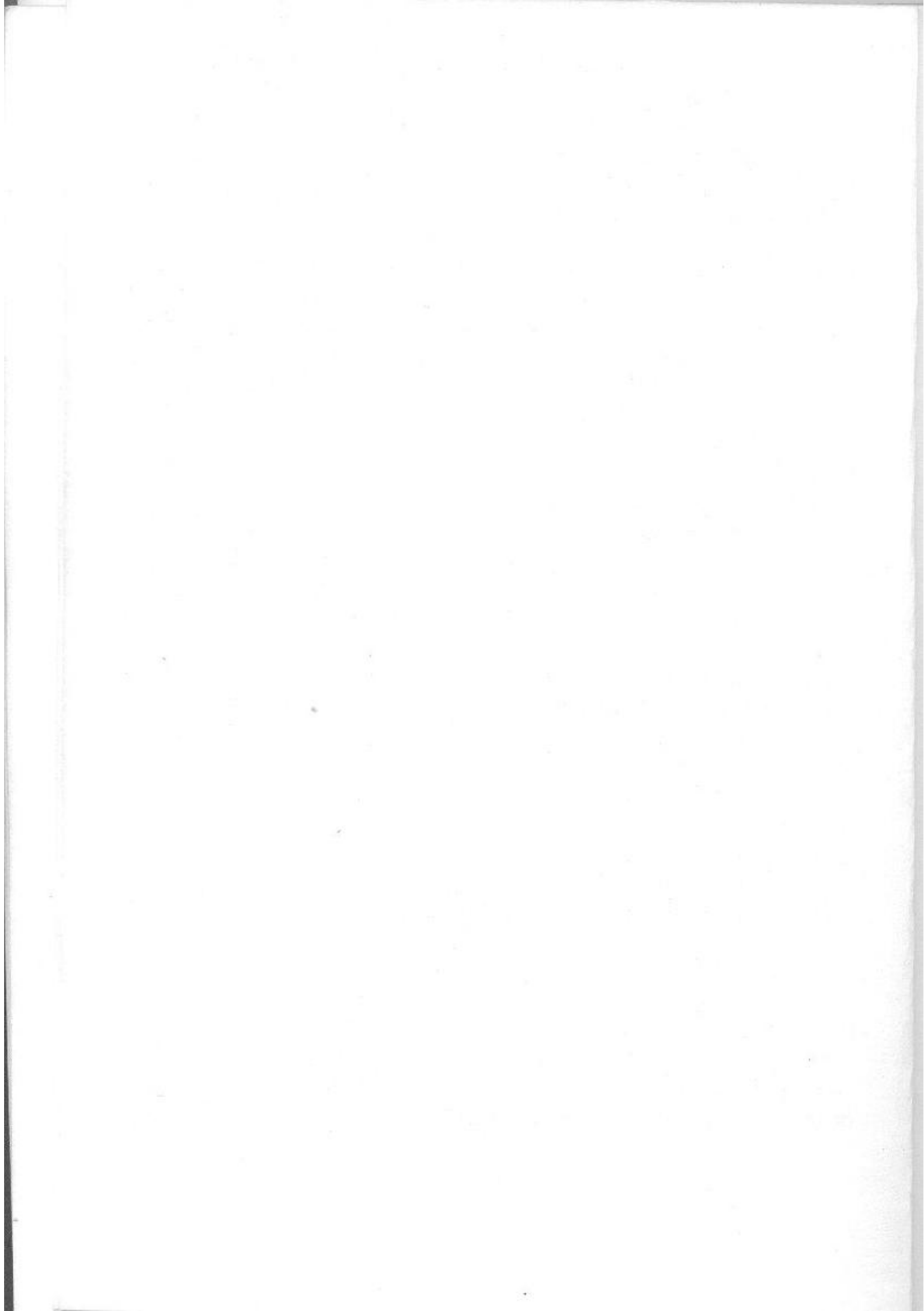
110100= 52D
-000111= 7D
101101= 45D

Numa subtração pode acontecer um resultado negativo, neste caso usa-se o bit mais à esquerda para se distinguir o sinal. Se o bit mais significativo (o bit mais à esquerda) é 1, então o número é negativo, caso contrário é positivo. Desta forma, o Z80 utiliza sete bits para armazenar o dado propriamente dito e um bit para indicar o sinal do mesmo.

Você deve ter notado que é muito mais simples realizar uma soma binária do que uma subtração. Os computadores utilizam um processo mais simples para realizar as subtrações, o chamado complemento a dois, que consiste em somar o complemento a dois do minuendo ao subtraendo. Desta forma o computador realiza a subtração como uma soma, dispensando assim o circuito de subtração.

A maneira de se obter o complemento a dois é invertendo-se cada um dos bits do número e somando-se 1 ao resultado. Conforme ficará claro examinando-se o exemplo seguinte:

	000111= 7D
invertendo-se os bits	111000
somando-se 1	111001
realizando a diferença	110100= 52D
	+111001= -7D
	101101= 45D

Com este exemplo concluímos o capítulo sobre a matemática nos computadores. A importância deste capítulo é grande portanto tenha certeza de que entendeu muito bem os conceitos aqui expostos antes de iniciar a leitura dos próximos capítulos.

CAPÍTULO 3

O Z80

Como já mencionei anteriormente a programação em linguagem de máquina envolve o conhecimento da máquina em si, ou melhor, do hardware. Este capítulo se destina a desvendar a maneira pela qual o Z80 executa suas tarefas.

O Z80 dispõe de 8 registros, conforme foi dito no capítulo 1, identificados a seguir:

A (acumulador)
F (registro dos flags)
B C
D E
H L

De todos os registros acima expostos, o mais privilegiado é o acumulador. Todas as operações de registros são mais rápidas quando usa-

mos o acumulador e todas as operações aritméticas e de lógica booleana usam o acumulador como operando e destino final.

O registro F é especial, pois é ele que informa ao sistema o resultado das operações aritméticas e lógicas realizadas. Esta informação está contida em 6 dos 8 bits disponíveis neste registro. Na tabela seguinte temos a designação de cada um desses bits (flags):

BIT	SÍMBOLO	NOME
0	C	CARRY
1	N	SUBTRACT
2	P/O	PARITY/OVERFLOW
3	-	NÃO USADO
4	A/C	AUXILIARY CARRY
5	-	NÃO USADO
6	Z	ZERO
7	S	SIGN

Abaixo, apresentamos um pequeno sumário das funções de cada um desses bits:

CARRY — Este bit indica a existência do vai " 1 " nas operações aritméticas. Nas operações booleanas este bit toma o valor " 0 ".

SUBTRACT — Este bit auxilia o Z80 na medida em que assume o valor "0" em todas as instruções de adição e o valor " 1 " em todas as instruções de subtração.

PARITY/OVERFLOW — Este bit exerce a função de dois flags num só. O flag de overflow assume o valor " 1 " quando o bit 7 é afetado pelo transporte do "vai 1 " do bit 6 em somas e subtrações. O flag de paridade indica a quantidade de bits ativos do re-

sultado. Se este número de bits for par este flag assume o valor " 1 ".

ZERO

Este flag assume o valor " 1 " quando as operações aritméticas e lógicas geram um resultado nulo, e assume o valor " 0 " quando essas operações geram um resultado não nulo.

SIGN

Quando um byte representa um número com sinal, o seu bit 7 armazena o sinal, se o bit 7 for " 1 " o número é negativo e se for " 0 " o número é positivo. Este flag reflete desta mesma forma o último resultado ocorrido.

AUXILIARY CARRY

Este flag é, na verdade, um carry auxiliar. Sua função é detectar o transporte do "vai 1" do bit 3 para o bit 4.

Vejamos agora, a função dos registros B e C. O registro B e o registro C, que faz par com ele, são usados basicamente como contadores. Já vimos anteriormente que o Z80 tem um flag que indica a ocorrência de resultados iguais a zero. Existe uma instrução que faz uso deste flag para transformar o registro B num contador de 8 bits, podendo portanto "contar" de 0 a 255. Esta instrução é a DJNZ, que quando executada inspeciona o registro B para ver se ele é igual a zero, se assim for o Z80 executa a próxima instrução, caso contrário decrementa o registro B e realiza um salto de instruções. Como vimos, esta instrução só manipula dados de 8 bits, mas já sabemos que os registros B e C unidos podem ser encarados como um registro de 16 bits. Para esta associação podemos contar com diversas instruções poderosas como por exemplo a instrução CPDR que instrui o Z80 a decrementar o par de registros BC de uma unidade, decrementar o conteúdo do par de registros HL, e comparar o conteúdo do registro A com o conteúdo da memória apontada pelo par HL.

Os registros D e E não possuem nenhuma função particular, podendo ser utilizados como memória temporária de acesso rápido, além de poderem armazenar posições de memória de interesse. São muito utilizados nas instruções de transferência de blocos(instruções LDIR e LDDR), onde o par DE é usado para indicar o destino da transferência.

O par de registros H e L é o privilegiado no grupo dos registros de 16 bits, podendo ser usado em operações de soma e subtrações de 16 bits, para guardar posições de memória úteis além de diversas outras operações de extrema importância, como ficará claro nos capítulos posteriores.

Além dos registros vistos acima, existem os chamados registros de índice, os pares de registros IX e IY, ambos de 16 bits. Estes registros não oferecem a possibilidade de desmembramento em registros de 8 bits, pelo menos se nos basearmos pela lista de mnemônicos fornecida pela ZILOG. Como afirmamos no primeiro capítulo o fato de não podermos desmembrar estes registros é mais teórico, pois na prática esta limitação não existe.

O registro PC é o indicador que o Z80 possui para indicar qual a instrução que está sendo executada. Este registro é incrementado automaticamente após a execução das instruções. O outro registro especial é o SP que guarda os endereços de retorno das subrotinas. Por esta razão, não devemos utilizá-lo para guardar dados, pois podemos alterar assim o endereço de retorno de uma subrotina e por conseqüência causar um **crash** no sistema. Um crash pode ser definido como uma perda de controle sobre o micro, que passa a realizar instruções diferentes das programadas.

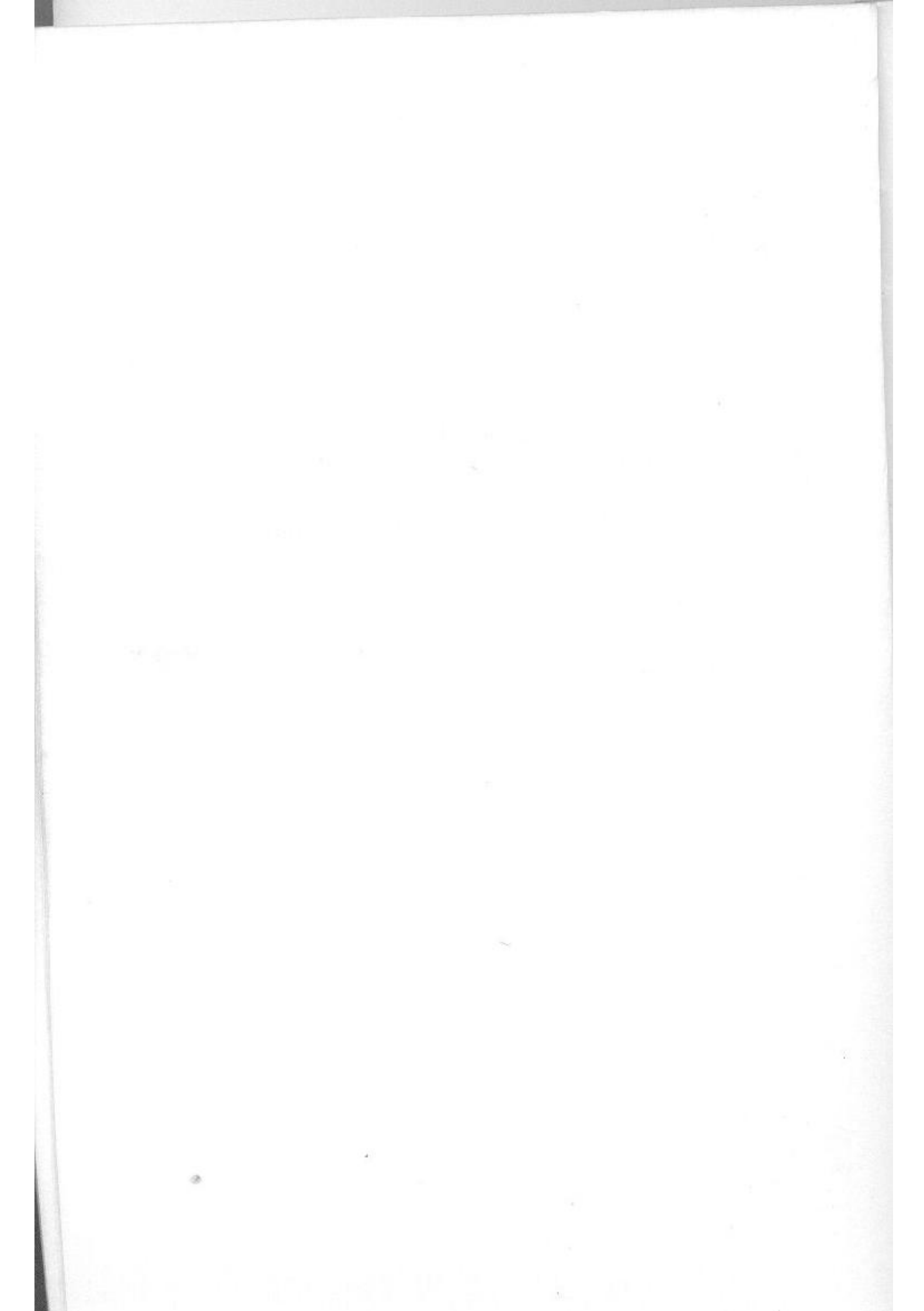
O Z80 possui ainda dois outros registros de uso muito limitado, são eles:

REGISTRO R É o registro responsável pelo refrescamento da memória. É muito usado para se obter números

aleatórios na faixa de 0 a 255.

REGISTRO I É o chamado vetor de interrupção. Este registro pode ser usado na proteção do software. Devido a sua complexidade vamos deixá-lo fora dos assuntos deste livro.

Se por acaso nos depararmos com uma falta de registros, podemos lançar mão dos chamados registros alternativos designados por: A', F', B', C', D', E', H' e L'. Infelizmente não podemos usar os registros alternativos simultaneamente com os registros "principais". A instrução EXX chaveia os grupos de registros, permitindo-nos passar de um conjunto para o outro.

Em cada um dos próximos capítulos iremos detalhar cada grupo de instruções do Z80, culminando após, com exemplos de como programar o MSX em linguagem de máquina.

CAPÍTULO 4

AS INSTRUÇÕES DE CARREGAMENTO

Neste capítulo iremos analisar o grupo de instruções mais utilizado pelo programador em linguagem de máquina. Os mnemônicos utilizados para denotar as instruções de carregamento começam todos com o prefixo LD, original da palavra inglesa LOAD (carregar). Esses tipos de instruções permitem colocar dados na memória ou nos registros, funcionando portanto, de modo semelhante as instruções POKE, PEEK e LET do BASIC do MSX. A apresentação desse grupo de instruções segue o padrão tradicional, ou seja, pelo tipo geral, pela função que realizam e os flags que afetam.

CARREGAMENTO DIRETO DOS REGISTROS DE 8 BITS

SINTAXE LD reg,dado

EXEMPLO LD A,03H
LD E,30H

FUNÇÃO Esta instrução é interpretada pelo Z80 como "coloque no registro especificado o valor do dado".

Depois de executar a instrução do primeiro exemplo o registro A conterá o valor 03H. Em BASIC podemos simular essa instrução pelo comando LET. Por exemplo: LET A=&H03.

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DIRETO DOS REGISTROS DE 16 BITS

SINTAXE LD par,dado de 16 bits

EXEMPLO LD HL,1000H
LD DE,90E9H

FUNÇÃO A função desta instrução é carregar nos pares de registros valores de 16 bits. No primeiro exemplo após a execução da instrução o registro H conterá o valor 10H e o registro L o valor 00H. Em BASIC podemos simular essa instrução com o comando LET. Por exemplo: LET HL=&H1000.

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DO ACUMULADOR COM DADOS DA MEMÓRIA

SINTAXE LD A,(end)

EXEMPLO LD A,(4000H)

FUNÇÃO Carregar o acumulador com um dado de uma determinada posição de memória. Suponhamos que o conteúdo da memória 4000H seja E9H, após a execução do exemplo acima, o acumulador conterá o valor E9H. Em BASIC podemos simular essa instrução com os comandos LET e PEEK. Por exemplo:
LET A=PEEK(&H4000)

FLAGS Não os afetados por essa instrução.

CARREGAMENTO DOS REGISTROS DE 16 BITS COM DADOS DE MEMÓRIA

SINTAXE LD par,(end)
LD indreg,(end)

EXEMPLO LD HL,(4000H)
LD IX,(5000H)

FUNÇÃO Carregar os pares de registros, tanto principais como indexados, com o conteúdo da memória especificada. No exemplo acima suponha que o endereço 4000H contenha o valor 10H e o endereço 4001H o valor 40H. Após a execução da instrução exemplificada o registro H conterá o valor 40H e o registro L o valor 10H. Podemos simular essa instrução em BASIC com comandos LET e PEEK. Por exemplo:
LET L=PEEK(&H4000): LET H=PEEK(&H4001)

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DA MEMÓRIA COM DADOS DE 8 BITS

SINTAXE LD (end),A

EXEMPLO LD (C000H),A

FUNÇÃO 0 Z80 interpreta essa instrução como "guarde o valor do registro A na memória endereçada por end". Se o registro A tiver o valor E9H então após à execução desta instrução a memória C000H irá conter o valor E9H. Em BASIC podemos simular essa instrução com o comando POKE. Por exemplo.
POKE &HC000,A

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DA MEMÓRIA COM DADOS DE 16 BITS

SINTAXE LD (end),par
LD (end),indreg

EXEMPLO LD (4000H),HL
LD (9000H),IX

FUNÇÃO Esta instrução ordena o Z80 a colocar na memória especificada por "end" o dado contido nos registros de 16 bits. Se por exemplo o registro H contém o valor A0H e o registro L o valor 00H, então após a execução da instrução a memória 4000H conterá o valor 00H e a memória 4001H o valor A0H. Em BASIC podemos simular essa instrução pelo comando POKE. Por exemplo:
POKE&H4000,L: POKE&H4001,H

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DE REGISTROS POR REGISTROS

SINTAXE LD reg,reg

EXEMPLO LD A,B

FUNÇÃO Esta instrução é interpretada como "guarde no registro A o valor contido no registro B". Embora nesse exemplo tenhamos usado só os registros A e B nada impede de usarmos os registros C, D, E, H e L nessas instruções. Em BASIC podemos simular esta instrução pelo comando LET. Por exemplo:
LET A=B

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DE REGISTRO POR ENDEREÇAMENTO INDIRETO

SINTAXE LD reg,(HL)
LD reg,(indreg+d)

EXEMPLO LD C,(HL)
LD E,(IX+5)

FUNÇÃO Essa instrução ordena o Z80 a guardar no registro "reg" o valor do conteúdo da memória endereçada por HL ou pelos registros indexados, sendo que nesse caso podemos contar com o recurso extra do incremento ou decremento "d". Em BASIC podemos simular essas instru-

ções pelos comandos PEEK e LET. Por exemplo:
LET C=PEEK(HL)

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DA MEMÓRIA POR ENDEREÇAMENTO INDIRETO

SINTAXE LD (HL),reg
LD (indreg+d),reg

EXEMPLO LD (HL),B
LD (IX+2),D

FUNÇÃO Essa instrução é interpretada como "guarde na memória endereçada por HL o valor contido no registro reg". Em BASIC podemos simular essa instrução pelo comando POKE. Por exemplo:
POKE HL,B.

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DIRETO DA MEMÓRIA POR ENDEREÇAMENTO INDIRETO

SINTAXE LD (HL),dado
LD (indreg+d),dado

EXEMPLO LD (HL),30H
LD (IX+4),A0H

FUNÇÃO Essa instrução carrega no endereço de memória endere-

çado por HL o dado especificado. Em BASIC podemos simular ess instrução pelo comando POKE. Por exemplo:
POKE HL,&H30

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DE REGISTROS POR ENDEREÇAMENTO INDIRETO

SINTAXE LD A,(par)
LD A,(indreg+d)

EXEMPLO LD A,(BC)
LD A,(IX+3)

FUNÇÃO Essa instrução ordena o Z80 a carregar o acumulador com o valor contido na memória endereçada pelos pares de registros. Em BASIC podemos simular essa instrução pelos comandos PEEK e LET. Por exemplo:
LET A=PEEK(BC)

FLAGS Não são afetados por essa instrução.

CARREGAMENTO DA MEMÓRIA POR REGISTROS

SINTAXE LD (par),A
LD (indreg+d),A

EXEMPLO LD (BC),A
LD (IX+0),A

FUNÇÃO Essa instrução é lida como "guarde na memória endereçada pelos registros de 16 bits, o valor contido no acumulador". Em BASIC podemos simular essa instrução pelo comandos POKE. Por exemplo:
POKE BC,A

FLAGS Não são afetadas por essa instrução.

MODIFICAÇÃO DO STACK POINTER

SINTAXE LD SP,dado de 16 bits
LD SP,par
LD SP,indreg

EXEMPLO LD SP,F380H
LD SP,HL
LD SP,IX

FUNÇÃO Essa instrução ordena o Z80 a colocar o valor de 16 bits fornecido, tanto diretamente, como por pares de registros de 16 bits, no Stack Pointer. Não existe similar dessa instrução em BASIC.

FLAGS Não são afetadas por essa instrução.

Neste capítulo vimos as principais instruções que permitem colocar um valor num registro ou numa posição de memória. Existem outras instruções especiais de carregamento que devido a sua complexidade só serão apresentadas num capítulo posterior.

CAPÍTULO 5

OPERAÇÕES ARITMÉTICAS

Como já vimos o Z80 é capaz de realizar operações aritméticas simples – adição e subtração – com valores inteiros de 8 e 16 bits. As operações de multiplicação e divisão são simuladas com **loops** de adições e subtrações, embora haja uma maneira mais inteligente de realizar estas operações mais complexas, conforme veremos num capítulo posterior. Ao examinar esse capítulo, você perceberá a utilidade do conjunto de flags do Z80. Suponha que você queira fazer uma adição de dois números de 8 bits e guardar o resultado num registro de 8 bits também. Suponha então, que os números são 190 e 210. Como você pode observar a soma é igual a 400, o que implica que este resultado não pode ser armazenado num registro de 8 bits, pois o maior valor que este pode armazenar é igual a 255. Quando ocorre uma situação semelhante a esta, a ULA (Unidade Lógica Aritmética) indica que houve um estouro (overflow) através do flag de OVERFLOW. Desta maneira saberemos que o resultado colocado no registro de 8 bits está errado. Por esse motivo (e por outros que veremos), tome muito cuidado ao utilizar estas instruções.

OPERAÇÕES ARITMÉTICAS DE 8 BITS

SINTAXE ADD A,dado

EXEMPLO ADD A,3FH

FUNÇÃO Esta instrução apenas soma o dado especificado, no caso do exemplo é 3FH, com o valor presente no registro A. Se A armazenar o valor 01H, após a execução da instrução acima passará a conter o valor 40H.

FLAGS Todas os flags são afetados com a exceção do flag N que indica a ocorrência de uma subtração.

SINTAXE ADD A,reg

EXEMPLO ADD A,B

FUNÇÃO Esta instrução é semelhante a anterior, só que agora o dado está contido num registro. Se A tiver o valor 10H e B o valor 40H, após a execução da instrução o registro A terá o valor 50H.

FLAGS Todos os flags são afetados com a exceção do flag N que indica a ocorrência de uma subtração.

SINTAXE ADD A,(HL)
ADD A,(indreg+d)

EXEMPLO ADD A,(HL)
ADD A,(IX+0)

FUNÇÃO Esta instrução faz com que o valor de 8 bits contido na

memória apontada por HL, IX, ou IY, seja somado ao valor presente no registro A. Suponha que HL=4000H, A=30H e que o conteúdo do endereço de memória 4000H seja 20H. Após a execução dessa instrução, o registro A conterá o valor 50H.

FLAGS Da mesma forma que as instruções anteriores, o único flag que não é afetado é o N.

SINTAXE ADC A,dado

EXEMPLO ADC A,30H

FUNÇÃO Esta instrução soma além do dado especificado, o valor do CARRY. Assim sendo, se no exemplo acima o registro A tiver o valor 10H e o CARRY estiver "ressetado" (valor igual a zero), após a execução o registro A terá o valor 40H. Se o CARRY estiver "setado" (valor igual a um), o registro A conterá o valor 41H. Uma maneira de se ressetar o CARRY sem modificar o conteúdo do registro A é utilizar a instrução AND A.

FLAGS Todos os flags são afetados com exceção do N.

SINTAXE ADC A,reg

EXEMPLO ADC A,B

FUNÇÃO O funcionamento dessa instrução é semelhante ao da anterior, só que agora o dado a ser somado encontra-se num registro de 8 bits.

FLAGS Todos os flags são afetados com exceção do N.

SINTAXE ADC A,(HL)
ADC A,(indreg+d)

EXEMPLO ADC A,(HL)
ADC A,(IX+1)

FUNÇÃO O funcionamento é igual ao das duas instruções anteriores, sendo que o dado a ser somado encontra-se na posição de memória endereçada por HL, IX, ou IY.

FLAGS Da mesma forma que as instruções anteriores o único flag não afetado é o N.

SINTAXE SUB dado

EXEMPLO SUB 20H

FUNÇÃO Esta instrução subtrai o valor especificado do valor contido no acumulador. Note que não se escreve SUB A,20H. Suponha que o valor do acumulador seja 30H, após a execução do exemplo acima o seu valor passará a ser 10H.

FLAGS Todos os flags são afetados.

SINTAXE SUB reg

EXEMPLO SUB B

FUNÇÃO Esta instrução é interpretada pelo Z80 como "subtraia do acumulador o valor do registro especificado".

FLAGS Todos os flags são afetados.

SINTAXE SUB (HL)
SUB (indreg+d)

EXEMPLO SUB (HL)
SUB (IX+0)

FUNÇÃO Esta instrução é semelhante as duas anteriores. O dado a ser subtraído está agora contido na posição de memória apontada por HL, IX ou IY.

FLAGS Todos os flags são afetados por essa instrução.

SINTAXE SBC A,dado

EXEMPLO SBC A,30H

FUNÇÃO Esta instrução ordena o Z80 a subtrair o dado especificado bem como o estado do flag do CARRY (0 ou 1), do valor atual do acumulador. No exemplo acima se A tiver o valor 31H e o CARRY estiver setado, após a execução o conteúdo do acumulador (registro A) passará a ser igual a zero.

FLAGS Todos os flags são afetados.

SINTAXE SBC A,reg

EXEMPLO SBC A,B

FUNÇÃO Esta instrução significa que o Z80 deve subtrair o valor do registro especificado (B no exemplo) e do estado do CARRY, do valor contido no acumulador.

FLAGS Todos os flags são afetados.

SINTAXE SBC A,(HL)
SBC A,(increg+d)

EXEMPLO SBC A,(HL)
SBC A,(IX+2)

FUNÇÃO Esta instrução funciona de modo semelhante ao das duas anteriores, só que agora o valor especificado está na posição de memória endereçada por HL, IX ou IY.

FLAGS Todos os flags são afetados.

SINTAXE INC reg

EXEMPLO INC A

FUNÇÃO Esta instrução incrementa em 1 o conteúdo do registro especificado. Se no exemplo acima o conteúdo do acumulador era 10H após a execução da instrução passou a ser 11H.

FLAGS Todos os flags são afetados com exceção do flag do CARRY.

SINTAXE DEC reg

EXEMPLO DEC A

FUNÇÃO Esta instrução decrementa em 1 o conteúdo do registro especificado. Se no exemplo acima o conteúdo do acumulador era 10H após a execução da instrução passou a ser 0FH.

FLAGS Todos os flags são afetados com exceção do flag do CARRY.

OPERAÇÕES ARITMÉTICAS DE 16 BITS

SINTAXE ADD HL, par

EXEMPLO ADD HL,HL

FUNÇÃO Esta instrução soma o conteúdo de um par de registros com o conteúdo do par HL. Se no exemplo acima HL=2000H após a execução teremos HL=4000H.

FLAGS Somente os flags do CARRY e SUBTRAÇÃO (C E N) são afetados.

SINTAXE ADD IX, par

EXEMPLO ADD IX,BC

FUNÇÃO Esta instrução soma o valor do registro IX com o valor dos pares BC, DE, IX ou SP. O funcionamento é semelhante ao da instrução anterior.

FLAGS Somente os flags do CARRY e SUBTRAÇÃO (C e N) são afetados.

SINTAXE ADD IY,par

EXEMPLO ADD IY,IY

FUNÇÃO Esta instrução funciona da mesma maneira que a anterior. Da mesma forma os pares que podem compor essa instrução são os seguintes: BC,DE,IY e SP.

FLAGS Somente os flags do CARRY e SUBTRAÇÃO (C e N) são afetados.

SINTAXE ADC HL,par

EXEMPLO ADC HL,BC

FUNÇÃO Esta instrução soma o conteúdo do par de registros especificado e o estado do CARRY ao par HL.

FLAGS Todos os flags são afetados.

SINTAXE SBC HL,par

EXEMPLO SBC HL,DE

FUNÇÃO Esta instrução subtrai o conteúdo do par de registros e o estado do CARRY ao conteúdo do par HL.

FLAGS Todos os flags são afetados.

SINTAXE INC par
INC indreg

EXEMPLO INC HL
INC IX

FUNÇÃO Esta instrução incrementa em 1 o conteúdo do par espe-

cificado.

FLAGS Os flags não são afetados por essa instrução.

SINTAXE INC (HL)
INC (indreg+d)

EXEMPLO INC (HL)
INC (IX+3)

FUNÇÃO Esta instrução incrementa em 1 o conteúdo da posição de memória endereçada por HL, IX ou IY.

FLAGS Somente o flag do CARRY não é afetado.

SINTAXE DEC par
DEC indreg

EXEMPLO DEC DE
DEC IY

FUNÇÃO Esta instrução decrementa em 1 o conteúdo do par de registros especificados.

FLAGS Não são afetados por essa instrução.

SINTAXE DEC (HL)
DEC (indreg+d)

EXEMPLO DEC (HL)
DEC (IX+4)

FUNÇÃO Esta instrução decrementa o conteúdo da posição de

memória apontada por HL, IX ou IY.

FLAGS Somente o flag do CARRY não é afetado.

Com este capítulo terminamos a apresentação do conjunto de instruções que realizam as operações aritméticas simples. No próximo capítulo iremos considerar as instruções responsáveis pelas operações lógicas e de comparação.

CAPÍTULO 6

INSTRUÇÕES LÓGICAS E DE COMPARAÇÃO

As instruções lógicas são, depois das instruções de carregamento, as mais usadas pelo programador em linguagem de máquina e até mesmo em linguagem BASIC, principalmente no teste de condições para tomada de decisões. Em BASIC o uso das instruções lógicas está quase sempre associado aos comandos IF, THEN e ELSE. Em linguagem de máquina a situação não é muito diferente, como você poderá perceber. As instruções lógicas podem ser divididas em quatro grupos, a saber: AND, OR, XOR e NOT.

AS INSTRUÇÕES DO TIPO AND

A operação lógica AND no Z80 nada mais faz, do que comparar bit a bit o conteúdo do acumulador com um registro, ou com um dado, ou ainda, com o conteúdo de uma posição de memória. A lógica envolvida numa operação AND está resumida na figura a seguir:

0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1

Como podemos observar o resultado de uma operação AND só será igual a 1 se ambos os bits envolvidos na comparação forem iguais a 1. As instruções possíveis são:

SINTAXE AND dado

EXEMPLO AND 0FH

FUNÇÃO Esta instrução realiza a operação AND entre o valor do acumulador e o dado especificado. Suponha que o acumulador contenha o valor 1FH (00011111B) e o dado especificado seja 0FH (00001111B). Após a execução da instrução AND 0FH, o acumulador apresentará o valor 0FH. Se você não se convenceu observe a figura seguinte:

```
     A = 1FH = 00011111B
DADO = 0FH = 00001111B
             ----------
 A AND 0FH  =00001111B = 0FH
```

FLAGS As instruções AND zeram o valor do CARRY e do flag de subtração (N) e, colocam em 1 o flag do CARRY AUXILIAR. Os demais flags (Z, P/O e S) passam a descrever o resultado da operação.

SINTAXE AND reg

EXEMPLO AND B

FUNÇÃO Esta instrução realiza a mesma operação que a anterior, só que agora, o dado especificado é o valor do registro.

FLAGS A situação dos flags após esta instrução é a mesma da instrução anterior.

SINTAXE AND (HL)
AND (indreg+d)

EXEMPLO AND (HL)
AND (IX+3)

FUNÇÃO Esta instrução opera do mesmo modo que as duas anteriores, sendo que nesta o dado está na posição de memória endereçada por HL, IX ou IY.

FLAGS A situação dos flags após a execução desta instrução é a mesma das duas instruções anteriores.

AS INSTRUÇÕES DO TIPO OR

Uma operação do tipo OR compara os bits de modo semelhante ao da instrução AND. A figura seguinte ilustra as quatro ocorrências possíveis:

0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1

Como se pode observar o resultaco só não será 1 se ambos os bits forem iguais a 0. As instruções possíveis são:

SINTAXE OR dado

EXEMPLO OR 0FH

FUNÇÃO Esta instrução pega o valor do acumulador e faz um OR com o dado especificado. Suponha que o acumulador contenha o valor 1FH. Após a execução da instrução o seu valor continuará sendo 1FH. Para entender este resultado observe a figura seguinte:

```
       A = 1FH  = 00011111B
    DADO = 0FH  = 00001111B
    ---------------------------
   A OR 0FH  = 00011111B = 1FH
```

FLAGS As instruções do tipo OR zeram o valor do CARRY e do flag de subtração (N), e, passam a 1, o valor do CARRY AUXILIAR. Os demais flags refletem o resultado da operação.

SINTAXE OR reg

EXEMPLO OR E

FUNÇÃO Esta instrução opera de modo semelhante à anterior, sendo que, neste caso, o dado especificado está armazenado num registro.

FLAGS Os flags são alterados da mesma forma que na instrução anterior.

SINTAXE OR (HL)
OR (indreg+d)

EXEMPLO OR (HL)
OR (IX+1)

FUNÇÃO Esta instrução atua do mesmo modo que as duas anteriores, sendo que agora, o dado a ser usado na operação OR, encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS A situação dos flags, após a execução desta instrução, é a mesma das duas instruções anteriores.

AS INSTRUÇÕES DO TIPO XOR

Estas instruções são úteis porque o resultado só assume o valor 1 se um dos bits é 1 e o outro é 0. A figura a seguir ilustra as quatro possíveis ocorrências:

0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0

As instruções que operam esta instrução são:

SINTAXE XOR dado

EXEMPLO XOR 0FH

FUNÇÃO Esta instrução realiza um XOR entre o conteúdo do acumulador e o dado especificado. Se o acumulador apresentar o valor 1FH e o dado for igual a 0FH, como no exemplo anterior, após a execução da instrução, o acumulador irá apresentar o valor 10H. Para melhor en-

tender, acompanhe a figura seguinte:

A = 1FH = 00011111B
DADO = 0FH = 00001111B
A XOR 0FH = 00010000B = 10H

FLAGS Esta instrução zera os flags do CARRY e de subtração (N), e coloca em 1 o valor do flag AC. Os demais flags refletem o resultado da operação.

SINTAXE XOR reg

EXEMPLO XOR B

FUNÇÃO Esta instrução realiza a operação XOR entre o valor do acumulador e o valor do registro especificado. A operação é idêntica à da instrução anterior.

FLAGS O estado dos flags ao final desta operação é o mesmo da operação anterior.

SINTAXE XOR (HL)
XOR (indreg+d)

EXEMPLO XOR (HL)
XOR (IX+0)

FUNÇÃO Esta instrução realiza a operação XOR entre o valor do acumulador e o valor da posição de memória endereçada por HL, IX ou IY.

FLAGS A situação dos flags ao final da operação é exatamente a mesma das duas instruções anteriores.

A INSTRUÇÃO DE COMPLEMENTO

Esta instrução realiza o complemento a um do valor contido no acumulador. Esta instrução é igual a instrução NOT do BASIC.

SINTAXE CPL

EXEMPLO CPL

FUNÇÃO Esta instrução realiza o complemento a 1 dos bits que compõe o valor armazenado no acumulador. Suponha que A contenha o valor 0FH, após a execução desta instrução o valor do acumulador passará a ser F0H. Para entender a afirmativa anterior acompanhe a figura seguinte:

A = 0FH = 00001111B
CPL = F0H = 11110000B

Como se pode ver a instrução CPL nada mais faz do que trocar o valor de cada bit do acumulador.

FLAGS Esta instrução só modifica os flags de subtração (N) e de CARRY AUXILIAR (A/C), colocando-os no valor 1.

AS INSTRUÇÕES DE COMPARAÇÃO

As instruções de comparação são muito usadas, pois são as responsáveis diretas pelas operações de comparação. O Z80 permite 3 variantes desta instrução, conforme se pode verificar a seguir:

SINTAXE CP dado

EXEMPLO CP 1AH

FUNÇÃO Esta instrução é interpretada como uma ordem para comparar o conteúdo do acumulador com o dado especificado. Suponha que o acumulador contenha o valor 20H e o dado especificado seja 1AH. A instrução realiza a comparação subtraindo o dado do valor do acumulador, sem contudo, alterar o valor deste último. A única alteração ocorre com os flags. Se o dado for igual ao conteúdo do acumulador o flag Z assume o valor 1, e o valor 0 no caso contrário. Se o valor do dado for menor ou igual ao valor do acumulador o flag C assume o valor 0, e o valor 1 no caso contrário. No caso do nosso exemplo o resultado seria Z=0, C=0 e A=20H.

FLAGS O flag de subtração (N) assume o valor 1 e os demais passam a refletir o resultado da operação de comparação.

SINTAXE CP reg

EXEMPLO CP B

FUNÇÃO Esta instrução atua da mesma maneira que a anterior sendo que agora o dado a ser comparado encontra-se num registro. É importante notar que estas instruções de comparação só alteram os flags.

FLAGS Os flags se alteram da mesma forma que na instrução anterior.

SINTAXE CP (HL)
CP (indreg+d)

EXEMPLO CP (HL)
CP (IX+2)

FUNÇÃO Esta instrução funciona do mesmo modo que as duas anteriores, sendo que agora o dado a ser comparado encontra-se na posição de memória indicada por HL, IX ou IY.

FLAGS Os flags são alterados da mesma forma que nas duas instruções anteriores.

Com este capítulo encerramos a apresentação das instruções de lógica booleana e de comparação, muito importantes nas tomadas de decisões, na movimentação de gráficos e na gravação de dados em k-7 ou disco.

CAPÍTULO 7

INSTRUÇÕES DE SALTO E LOOPS

Em quase todas as linguagens de alto nível, temos algum tipo de instrução que permita alterar a seqüência normal do programa através de desvios. No BASIC temos os comandos GOSUB e GOTO. Em assembly (linguagem de máquina) temos os comandos JR e JP semelhantes ao GOTO do BASIC, e o comando CALL que atua de modo semelhante ao GOSUB.

AS INSTRUÇÕES DE SALTO

Como já dissemos este grupo se compõe de duas instruções. JR e JP. A instrução JR significa salto relativo, do inglês **jump relative**. A vantagem de usarmos esta instrução se deve a duas razões principais:

1. Só gastar dois bytes;
2. Por representar saltos relativos, permite que o programa seja

relocável, isto é, funcione em áreas de memória diferentes da programada, sem necessidade de alterações.

A única desvantagem desta instrução está no fato dos saltos estarem limitados a 129 bytes para a frente e 126 para trás. Usamos o termo byte em vez de posições de memória, por ser esta a designação mais usada.

A instrução JP realiza um salto absoluto, ou seja, para uma determinada posição de memória específica. A única desvantagem esta no fato de gastar 3 bytes (com a exceção de JP (HL) e JP (indreg) que gastam só 1 byte e 2 bytes respectivamente). As vantagens são:

1. O salto não é limitado;
2. A execução é mais rápida, pois não se calculam os endereços para o desvio

Abaixo segue uma lista contendo as instruções possíveis em assembly do grupo analisado acima.

SINTAXE JR d

EXEMPLO JR 32H

FUNÇÃO Esta instrução realiza um salto relativo de tantas posições de memória quantas forem indicadas pelo byte d. No caso do exemplo a instrução provoca um salto de 32H bytes.

FLAGS Não são afetados por esta instrução.

SINTAXE JR cond,d

EXEMPLO JR C,80H

FUNÇÃO Esta instrução provoca um desvio na execução do programa se uma determinada condição tiver ocorrido. No caso do exemplo se o CARRY estiver setado a instrução provocará um salto de 128 bytes. As condições podem ser C, NC, Z e NZ.

FLAGS Não são afetados por esta instrução.

SINTAXE JP end

EXEMPLO JP 4000H

FUNÇÃO Esta instrução ordena o desvio do programa para a posição de memória endereçada por end. No caso do exemplo o registro PC irá se alterar para o valor 4000H provocando o desvio do programa para esta posição de memória.

FLAGS Não são alteracos por esta instrução.

SINTAXE JP (HL)
JP (indreg)

EXEMPLO JP (HL)
JP (IX)

FUNÇÃO Esta instrução desvia o programa para a posição de memória apontada por HL, IX ou IY.

FLAGS Não são afetados por esta instrução.

SINTAXE JP cond,end

EXEMPLO JP Z,4000H

FUNÇÃO Esta instrução realiza o desvio para uma determinada posição de memória somente se a condição especificada ocorrer. As condições podem ser C, NC, Z, NZ, P, M, PO e PE.

FLAGS Não são afetados por esta instrução.

A instrução CALL funciona de modo semelhante ao comando GOSUB do BASIC. Ao encontrar uma instrução CALL executável (pode ocorrer uma instrução condicional), o Z80 guarda o endereço da próxima instrução no SP e salta para o endereço da sub-rotina. A saída da sub-rotina é realizada pela instrução RET, que também pode ser condicional. Ao encontar uma instrução RET executável o Z80 pega o valor contido no SP e realiza um salto para o endereço especificado pelo valor pego do SP. É por este motivo que antes de realizar uma instrução RET devemos tomar o cuidado de não alterar o conteúdo do SP. Como você pode notar uma sub-rotina envolve um planejamento cuidadoso para que não ocorram desvios indesejáveis na execução de um programa.

SINTAXE CALL end

EXEMPLO CALL 5A00H

FUNÇÃO Esta instrução ordena o Z80 a chamar uma sub-rotina que se inicia no endereço especificado por end. No exemplo acima o Z80 irá chamar uma sub-rotina que se inicia no endereço 5A00H.

FLAGS Não são alterados por esta instrução.

SINTAXE CALL cond,end

EXEMPLO CALL C,4F00H

FUNÇÃO Esta instrução ordena o Z80 a chamar a sub-rotina que se inicia em end, caso a condição seja verdadeira. A condição pode ser C, NC, Z, NZ, P, M, PO e PE.

FLAGS Não são alterados por esta instrução.

SINTAXE RET

EXEMPLO RET

FUNÇÃO Esta instrução ordena o Z80 a executar a instrução imediatamente após aquela que provocou o desvio para a sub-rotina. Em outras palavras esta instrução realiza o retorno da sub-rotina.

FLAGS Não são alterados por esta instrução.

SINTAXE RET cond

EXEMPLO RET C

FUNÇÃO Esta instrução provoca o retorno de uma sub-rotina se uma determinada condição tiver ocorrido. As condições possíveis são: C, NC, Z, NZ, P, M, PO e PE.

AS INSTRUÇÕES DE LOOP

Um loop nada mais é que uma repetição de um mesmo comando por um número determinado de vezes. Observe o trecho de programa em BASIC a seguir.

```
10 FOR A=0 T0 255
20 PRINT CHR$(A);
30 NEXT A
```

No programa acima fizemos um loop com 256 (0 a 255) iterações. Em assembly podemos realizar um loop que funcione de modo semelhante a estrutura FOR/NEXT do BASIC. A instrução DJNZ realiza um loop de modo semelhante ao FOR/NEXT. Esta instrução usa o registro B como contador. Cada vez que o Z80 a executa acontece o seguinte:

1. Decrementa B;
2. Se o conteúdo de B não for zero salta para o início do loop, que estará entre −126 a +129 bytes desta instrução, e;
3. Se o conteúdo de B for zero, a instrução seguinte será realizada.

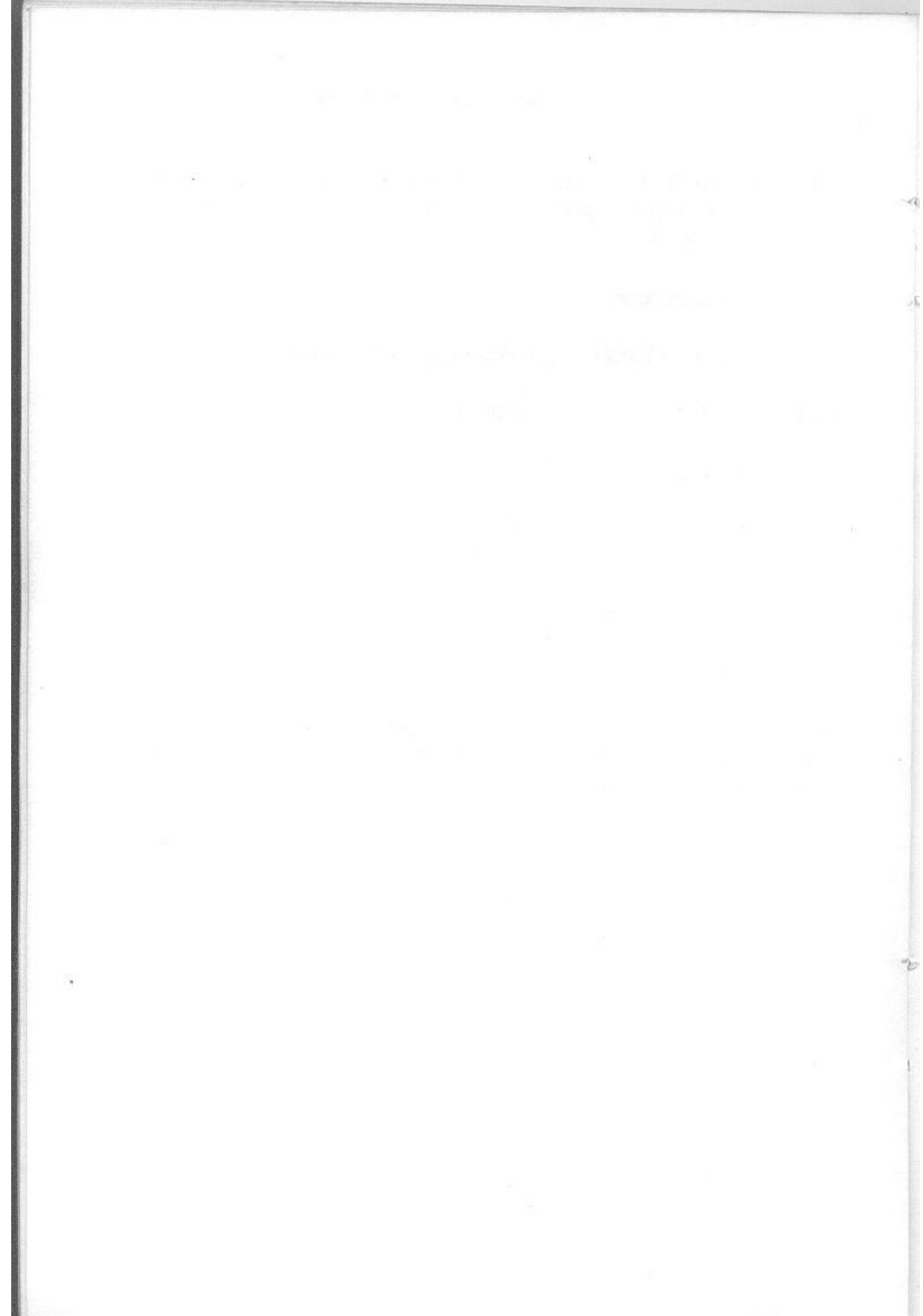
Seja o mesmo programa em BASIC escrito em assembly:

```
        ORG 0E000H     ;define o início do programa na memória

        LD B,00H       ;prepara B para loop de 256 iterações

        XOR A          ;zera o acumulador
LOOP    CALL 00A2H     ;chama sub-rotina na ROM que imprime o
                       ;caracter contido em A

        INC A          ;A=A+1

        DJNZ LOOP      ;se B diferente de zero imprime o próximo
                       ;caracter

        RET            ;retorna
```

Como você pode notar a principal limitação do comando DJNZ é o fato de só poder realizar loops de 256 iterações no máximo. Esta limitação

pode ser resolvida se usarmos o par de registros BC. Na figura a seguir temos um pequeno exemplo de como utilizar o par de registros BC como contadores.

```
        ORG 0E000H

        LD BC,50000D   ; prepará BC para 50000 iterações

LOOP    DEC BC         ; BC=BC-1

        LD A,B         ; A=B

        OR C           ; se B e C iguais a zero então o flag Z se-
                       ; rá setado

        JR NZ,LOOP     ; se BC é diferente de 0 volta

        RET            ; retorna
```

Com este capítulo você aprendeu a lidar com desvios e a realizar iterações simples. Nos próximos capítulos veremos como realizar operações de transferências de blocos e manipulação de bits.

CAPÍTULO 8

TRANSFERÊNCIA E PESQUISA DE BLOCOS

Por diversas vezes ocorre a necessidade de movermos um determinado bloco de bytes de uma posição de memória para uma outra posição. Em BASIC uma transferência envolve pelo menos a utilização dos seguintes comandos: LET, PEEK, POKE, FOR e NEXT. Em linguagem de máquina, a situação se resume na utilização de um único comando e alguns parâmetros, como podemos verificar a seguir.

SINTAXE LDD

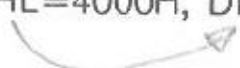
EXEMPLO LDD

FUNÇÃO Esta instrução transfere o conteúdo da posição de memória endereçada por HL para a posição de memória endereçada por DE. Além disso decrementa o conteúdo dos pares HL, DE e BC. Por exemplo, suponha que HL=4000H, DE=9000H, BC=3002H e que o conteúdo

da memória 4000H seja A0H. Após a execução da instrução HL=3FFFH, DE=8FFFH, BC=3001H e a posição de memória 9000H conterá agora o valor A0H.

FLAGS Os flags de AC e N são ressetados (iguais a zero), e o flag P/C é setado (igual a 1) se BC for diferente de zero, e é ressetado (igual a zero) se BC for igual a zero.

SINTAXE LDDR

EXEMPLO LDDR

FUNÇÃO Esta função atua do mesmo modo que a anterior com a execeção de só terminar a transferência quando BC for igual a zero. Enquanto a instrução anterior transfere somente um byte da posição de memória endereçada por HL para a posição endereçada por DE, esta instrução transfere tantos bytes quantos forem indicados por BC. Se por exemplo BC=4000H, a instrução LDDR irá tranferir 4000H bytes.

FLAGS São afetados da mesma forma que na instrução anterior.

SINTAXE LDI

EXEMPLO LDI

FUNÇÃO Esta instrução realiza a transferência de um byte da posição de memória endereçada por HL para a posição de memória endereçada por DE. A única diferença em relação a instrução LDD é que nessa os registros HL e DE são incrementados e BC é decrementado. Se por exemplo HL=4000H, DE=9000H, BC=3002H e a posição de

memória 4000H contiver o dado A0H, após a execução da Instrução LDI teremos: HL=4001H, DE=9001H, BC=3001H e a posição de memória 9000H passará a conter o dado A0H.

FLAGS Os flags são afetados exatamente da mesma maneira que nas instruções anteriores.

SINTAXE LDIR

EXEMPLO LDIR

FUNÇÃO Esta instrução atua de modo semelhante à anterior sendo que agora o número de bytes transferidos é dado pelo conteúdo do par de registros BC. Assim se BC for igual a 2000H, a quantidade de bytes transferidos será igual a 8192D.

FLAGS Os flags são afetados exatamente da mesma maneira que nas instruções anteriores.

SINTAXE CPI

EXEMPLO CPI

FUNÇÃO Esta instrução realiza uma comparação entre o conteúdo do acumulador e o conteúdo da posição de memória endereçada por HL. Logo após, incrementa o par HL e decrementa o par BC. Suponha que HL=4000H, A=C9H, BC=0001H e que a posição de memória indicada por HL contenha o valor C9H. Após a execução desta instrução teremos HL=4001H, A=C9H, BC=0000H e o flag Z=1, pois o conteúdo da memória é igual ao do acumulador.

FLAGS Os flags S, AC, P/O e Z passam a refletir o resultado da operação, já o flag C não é afetado e o flag N é setado (igual a 1).

SINTAXE CPIR

EXEMPLO CPIR

FUNÇÃO Esta instrução realiza a mesma função que a anterior sendo que agora a comparação se estende a um bloco de memória com início indicado por HL e tamanho indicado por BC. O processo de comparação do conteúdo do acumulador com o valor da memória endereçada por HL se repete até A=(HL), ou até que o par BC assuma o valor zero. Da mesma forma que a instrução anterior, HL é incrementado e BC decrementado.

FLAGS Os flags são afetados da mesma forma que na instrução anterior.

SINTAXE CPD

EXEMPLO CPD

FUNÇÃO Esta instrução realiza a comparação do conteúdo do valor do acumulador com o valor da posição de memória endereçada por HL. A única diferença em relação à instrução CPI é que agora os pares HL e BC são decrementados. Suponha por exemplo que o par HL=4000H, BC=0001H, A=E2H e que a posição de memória 4000H contenha o valor A0H. Após a execução desta instrução teremos: HL=3FFFH, BC=0000H, A=E2H e o flag Z=0, pois o conteúdo da posição de memória é diferente do valor do acumulador.

FLAGS São afetados da mesma forma que nas duas instruções anteriores.

SINTAXE CPDR

EXEMPLO CPDR

FUNÇÃO Esta instrução realiza o mesmo tipo de comparação que a instrução anterior sendo que agora o processo é repetido até A=(HL) ou até que BC=0. Da mesma forma que na instrução anterior os pares HL e BC são decrementados.

FLAGS São afetados da mesma forma que nas três instruções anteriores.

Com este capítulo terminamos a apresentação de um grupo de instruções extremamente poderoso, uma vez que permite o deslocamento de blocos inteiros de bytes ou a pesquisa da ocorrência de um determinado valor dentro de um bloco específico de bytes. No próximo capítulo apresentaremos o grupo de instruções que executam manipulações com o STACK e com os registros alternativos.

CAPÍTULO 9

AS OPERAÇÕES DE STACK E TROCAS DE REGISTROS

Como já tivemos oportunidade de mencionar, o **STACK** (pilha) nada mais é do que um armazém de endereços e portanto manipula unidades de informação de 16 bits. No MSX o STACK é posicionado no endereço F0A0H no sistema em k-7, e na posição E19FH no sistema que contenha um único drive. Outra característica importante, é o fato do STACK crescer para baixo à medida que vão sendo armazenados os dados, isto quer dizer que, se na versão em k-7 armazenarmos um endereço na pilha a nova posição do STACK passará a ser F09EH (F0A0H-2H). Uma outra característica já mencionada anteriormente é a de que as operações de STACK obedecem a regra LIFO (primeiro a entrar é o último a sair), que se não for seguida poderá levar o seu computador a realizar operações diferentes das desejadas, pois não se esqueça que o STACK é automaticamente usado por algumas instruções, sobretudo a instrução CALL.

As instruções que atuam diretamente sobre o STACK são as seguintes:

SINTAXE PUSH par
PUSH indreg

EXEMPLO PUSH HL
PUSH IX

FUNÇÃO Esta função ordena o Z80 a colocar o conteúdo do par de registros especificado (AF, BC, DE, HL, IX ou IY) na pilha (STACK).

FLAGS Não são afetados por esta instrução

SINTAXE POP par
POP indreg

EXEMPLO POP BC
POP IY

FUNÇÃO Esta instrução coloca no registro especificado o dado contido na pilha, ou seja realiza a operação inversa da instrução anterior. Como você já sabe, não existe a instrução LD par, par, e uma maneira de contornarmos este problema é usando estas duas últimas instruções. Suponha que você queira tornar BC igual HL, para tanto bastaria dar um PUSH HL seguido de um POP BC.

FLAGS Não são afetados por esta instrução.

SINTAXE EX (SP),HL
EX (SP),indreg

EXEMPLO EX (SP),HL
EX (SP),IX

FUNÇÃO Esta instrução ordena o Z80 a trocar o conteúdo do topo do STACK com o conteúdo de HL ou IX ou ainda IY. Note que só muda o valor contido no topo do STACK, os valores dos registros continuam inalterados.

FLAGS Não são alterados por esta instrução.

Você agora já sabe como manipular o conteúdo do STACK mas aqui vai novamente um aviso para que você tome todo o cuidado em dar o mesmo número de POP's e PUSH's. De outra forma você não poderá fazer um PUSH seguido de um CALL, para passar dados através do STACK para uma sub-rotina, pois o primeiro POP que você der dentro da sub-rotina irá recuperar não o valor pretendido, mas sim o endereço de retorno da sub-rotina.

Às vezes, há necessidade de se utilizar mais registros além dos disponíveis, ou então de se preservar os atuais flags, ou ainda, de se preservar todos os registros. Para este fim o Z80 possui as chamadas instruções de trocas. São elas:

SINTAXE EX AF,AF'

EXEMPLO EX AF,AF'

FUNÇÃO Esta instrução quando executada troca o conteúdo do par AF com o conteúdo do seu alternativo AF'. Esta instrução é particularmente útil quando desejamos preservar o estado atual dos flags.

FLAGS São trocados por esta instrução.

SINTAXE EXX

EXEMPLO EXX

FUNÇÃO Esta instrução instrui o Z80 a trocar o conteúdo dos pares BC, DE e HL pelos seus alternativos correspondentes. Desta forma com uma única instrução preservamos todos os registros.

FLAGS Não são afetados por esta instrução.

SINTAXE EX DE,HL

EXEMPLO EX DE,HL

FUNÇÃO Esta instrução troca o conteúdo de DE com o conteúdo de HL. Se DE=4000H e HL=A000H, após a execução desta instrução teremos HL=4000H e DE=A000H.

FLAGS Não são afetados por esta instrução.

Com este capítulo aprendemos a manipular as instruções de STACK e as instruções de troca, que como vimos são ambas muito poderosas. Nos dois próximos capítulos iremos aprender as instruções que manipulam os bits que constituem os valores dos registros e das posições de memória. Estes tipos de instruções são bastante usados em jogos como você poderá observar.

CAPÍTULO 10

MANIPULAÇÃO DE BITS

Por vezes torna-se necessário saber se um determinado bit de um registro ou posição de memória, está setado ou não, como por exemplo na leitura do teclado e/ou joysticks no processamento de um jogo. Como você poderá observar num capítulo posterior, muitas informações sobre o estado da máquina são fornecidas a nível de bits, entre elas podemos destacar a leitura e gravação de dados em K-7, a leitura dos joysticks e do teclado e o acionamento ou não do click do teclado. As instruções que manipulam os bits são as seguintes:

SINTAXE BIT b,reg

EXEMPLO BIT 4,A

FUNÇÃO Esta instrução realiza o teste para ver se o BIT especificado está setado ou não. Se o bit estiver setado o flag Z será igual a zero e igual a um em caso contrário. Como

se pode verificar, esta instrução coloca no flag Z o complemento do bit analisado. Note que na sintaxe o argumento b só pode assumir valores que vão de 0 a 7. Para exemplificar, suponha que A=11011110B, então BIT 4,A colocará o flag Z igual a zero, já a instrução BIT 5,A colocará o flag Z igual a um.

FLAGS O flag Z reflete o resultado da operação, o flag C não é alterado, o flag AC é colocado em 1, o flag N é colocado em zero e os flags S e P/O podem assumir qualquer valor.

SINTAXE BIT b,(HL)
BIT b,(indreg+d)

EXEMPLO BIT 5,(HL)
BIT 2,(IX+1)

FUNÇÃO Esta instrução realiza exatamente a mesma função que a instrução anterior, sendo que agora o bit a ser testado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS São afetados da mesma forma que na instrução anterior.

SINTAXE SET b,reg

EXEMPLO SET 3,A

FUNÇÃO Esta instrução coloca em 1 o bit b do registro reg. Se por exemplo A=11000101B, a instrução SET 3,A fará A=11001101B.

FLAGS Não são afetados por esta instrução.

SINTAXE SET b,(HL)
SET b,(indreg+d)

EXEMPLO SET 3,(HL)
SET 2,(IX+0)

FUNÇÃO Esta instrução atua do mesmo modo que a instrução anterior, sendo que agora o bit a ser setado encontra-se numa posição de memória endereçada por HL, IX ou IY, e não num registro.

FLAGS Não são afetados por esta instrução.

SINTAXE RES b,reg

EXEMPLO RES 6,B

FUNÇÃO Esta instrução realiza o inverso das duas instruções anteriores, ou seja, resseta o valor do bit b do registro reg. Suponha que B=11000001B, a instrução RES 6,B fará B=10000001B.

FLAGS Não são afetados por esta instrução.

SINTAXE RES b,(HL)
RES b,(indreg+d)

EXEMPLO RES 3,(HL)
RES 2,(IX+3)

FUNÇÃO Esta instrução realiza a mesma função que a instrução anterior, sendo que agora o bit a ser ressetado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS Não são afetados por esta instrução.

No próximo capítulo iremos aprender um grupo ainda mais extenso e bastante usado, o das rotações.

CAPÍTULO 11

ROTAÇÕES DE BITS

Estas instruções apesar de poderosas são pouco usadas de uma forma geral. No MSX no entanto, podemos observá-las em sub-rotinas que manipulam o PPI, em particular naquelas que habilitam as páginas inferiores da RAM. O termo página vem da capacidade do PPI de chavear blocos de 16Kb, sendo assim, se atribui o nome de página a um bloco de 16Kb de memória (ROM ou RAM). Desta forma temos a página zero que representa a memória compreendida entre os endereços 0000H e 3FFFH, a página um que compreende a memória do endereço 4000H ao 7FFFH, a página dois que compreende a memória do endereço 8000H ao BFFFH e a página três que compreende a memória entre os endereços C000H e FFFFH. Estes conceitos serão ampliados no capítulo sobre o PPI, por ora basta ter em mente que a utilização das instruções de rotação encontra no PPI uma boa aplicação. Abaixo descrevemos as instruções de rotação no nosso formato tradicional.

SINTAXE RLCA

EXEMPLO RLCA

FUNÇÃO Esta instrução realiza uma rotação circular para a esquerda passando o conteúdo do bit 7 para o CARRY e para o bit 0. Suponha que A=01010110B, ao executarmos esta instrução teremos A=10101100B e o flag C=0.

FLAGS O flag C é modificado como descrito acima, os flags AC e N são colocados em zero, os flags Z e S passam a representar o resultado final e os demais flags não são alterados.

SINTAXE RLC reg

EXEMPLO RLC B

FUNÇÃO Esta instrução atua do mesmo moco que a anterior sendo que agora o deslocamento circular a esquerda será realizado sobre o conteúdo do registro reg que é diferente de A.

FLAGS Os flags C, Z e S passam a refletir o resultado da operação, os flags AC e N são colocados em zero e o flag P/O é colqcado em P.

SINTAXE RLC (HL)
RLC (indreg+d)

EXEMPLO RLC (HL)
RLC (IX+2)

FUNÇÃO Esta instrução atua do mesmo modo que as duas ante-

riores, sendo que agora o dado a ser rotacionado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS São afetados da mesma forma que na instrução anterior.

SINTAXE RLA

EXEMPLO RLA

FUNÇÃO Esta instrução realiza uma rotação completa para a esquerda, sendo que o conteúdo do bit 7 vai para o CARRY e o conteúdo do CARRY vai para o bit 0 do acumulador. Suponha que A=01111011B e C=1, após a execução da instrução teremos A=11110111B e C=0 como poderíamos prever.

FLAGS O flag C é alterado da forma vista anteriormente, os flags AC e N são colocados em zero e os demais não são alterados.

SINTAXE RL reg

EXEMPLO RL B

FUNÇÃO Esta instrução atua do mesmo modo que a anterior, só que agora a rotação se faz sobre os outros registros de 8 bits diferentes de A.

FLAGS Os flags C, Z e S são alterados de acordo com o resultado da operação, o flag P/O ativa P e os demais são colocados em zero.

SINTAXE RL (HL)
RL (indreg+d)

EXEMPLO RL (HL)
RL (IY+2)

FUNÇÃO Esta instrução opera do mesmo modo que as duas anteriores, sendo que agora o dado a ser rotacionado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS São alterados da mesma forma que na instrução anterior.

SINTAXE RRCA

EXEMPLO RRCA

FUNÇÃO Esta instrução realiza o inverso da RLCA, pois roda para a direita transferindo o conteúdo do bit zero para o CARRY e para o bit 7 do acumulador. Suponha que A=10001010B e C=1, após a execução teremos A=01000101B e C=0.

FLAGS O flag C é alterado, os flags AC e N são colocados em zero e os demais não são alterados.

SINTAXE RRC reg

EXEMPLO RRC B

FUNÇÃO Esta instrução realiza a mesma operação da instrução anterior, sendo que agora o dado a ser rotacionado en-

contra-se nos demais registros de 8 bits diferentes do acumulador.

FLAGS Os flags C, Z e S passam a refletir o resultado da operação, os flags AC e N são colocados em zero e o flag P/O é colocado em P.

SINTAXE RRC (HL)
RRC (indreg +d)

EXEMPLO RRC (HL)
RRC (IX+0)

FUNÇÃO Esta instrução realiza exatamente a mesma operação da instrução anterior, sendo que agora o dado a ser rotacionado se encontra na posição de memória endereçada por HL, IX ou IY.

FLAGS São alterados da mesma forma que na instrução anterior.

SINTAXE RRA

EXEMPLO RRA

FUNÇÃO Esta instrução roda o conteúdo do acumulador para a direita passando o conteúdo co bit zero para o CARRY e o conteúdo prévio deste para o bit 7 do acumulador. Suponha A=11110010B e C=1, após execução teremos A=11111001B e C=0.

FLAGS O flag C é alterado, os flags AC e N são ressetados e os demais não são alterados.

SINTAXE RR reg

EXEMPLO RR D

FUNÇÃO Esta instrução realiza a mesma operação que a anterior, sendo que agora o dado a ser rotacionado encontra-se no registro especificado diferente do acumulador.

FLAGS Os flags C, Z e S passam a refletir o resultado da operação, os flags AC e N são ressetados e o flag P/O é colocado em P.

SINTAXE RR (HL)
RR (indreg+d)

EXEMPLO RR (HL)
RR (IX+2)

FUNÇÃO Esta instrução realiza a mesma operação das duas anteriores, sendo que o dado a ser rotacionado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS São alterados da mesma forma que na instrução anterior.

SINTAXE SLA reg

EXEMPLO SLA B

FUNÇÃO Esta instrução realiza um deslocamento para a esquerda, transportando o bit 7 para o CARRY e ressetando o bit 0 do registro. Suponha que o registro A seja igual a 01111011B, a instrução SLA A fará com que A passe a

ser 11110110B e C=0. Note que neste capítulo temos usado C para designar o flag do CARRY e não o registro C. Esta instrução é bastante útil porque na prática o que ela efetua é uma multiplicação por 2 sobre o valor do registro reg.

FLAGS Os flags C, Z, S e P/O passam a representar o resultado da operação e os flags AC e N são ressetados.

SINTAXE SLA (HL)
SLA (indreg+d)

EXEMPLO SLA (HL)
SLA (IX+20H)

FUNÇÃO Esta instrução realiza a mesma operação que a anterior com a exceção de que o dado agora está presente numa posição de memória e não num registro.

FLAGS São afetados da mesma forma que na instrução anterior com a exceção do flag P/O que é colocado em P.

SINTAXE SRA reg

EXEMPLO SRA A

FUNÇÃO Esta instrução desloca um bit para a direita, o conteúdo do registro reg, preservando o bit 7. Se A=10100011B, após a execução teríamos o acumulador igual a 11010001B e CARRY=1, pois o bit 0 é transferido para o CARRY. Na prática esta instrução realiza uma divisão por 2 do registro especificado.

FLAGS Os flags C, Z e S representam o resultado da operação, o flag P/O é colocado em P e os demais são ressetados.

SINTAXE SRA (HL)
SRA (indreg+d)

EXEMPLO SRA (HL)
SRA (IX+2AH)

FUNÇÃO Esta instrução realiza a mesma operação que a anterior, sendo que agora o dado a ser deslocado encontra-se na posição de memória endereçada por HL, IX ou IY.

FLAGS São afetados da mesma forma que pela instrução anterior.

SINTAXE SRL reg

EXEMPLO SRL D

FUNÇÃO Esta instrução realiza o deslocamento para a direita sem preservar o bit 7, pois passa o conteúdo do bit 0 para o CARRY e para o bit 7. Na prática também realiza uma divisão por 2 só que perdendo o bit do sinal (bit 7).

FLAGS São afetados da mesma forma que nas instruções anteriores

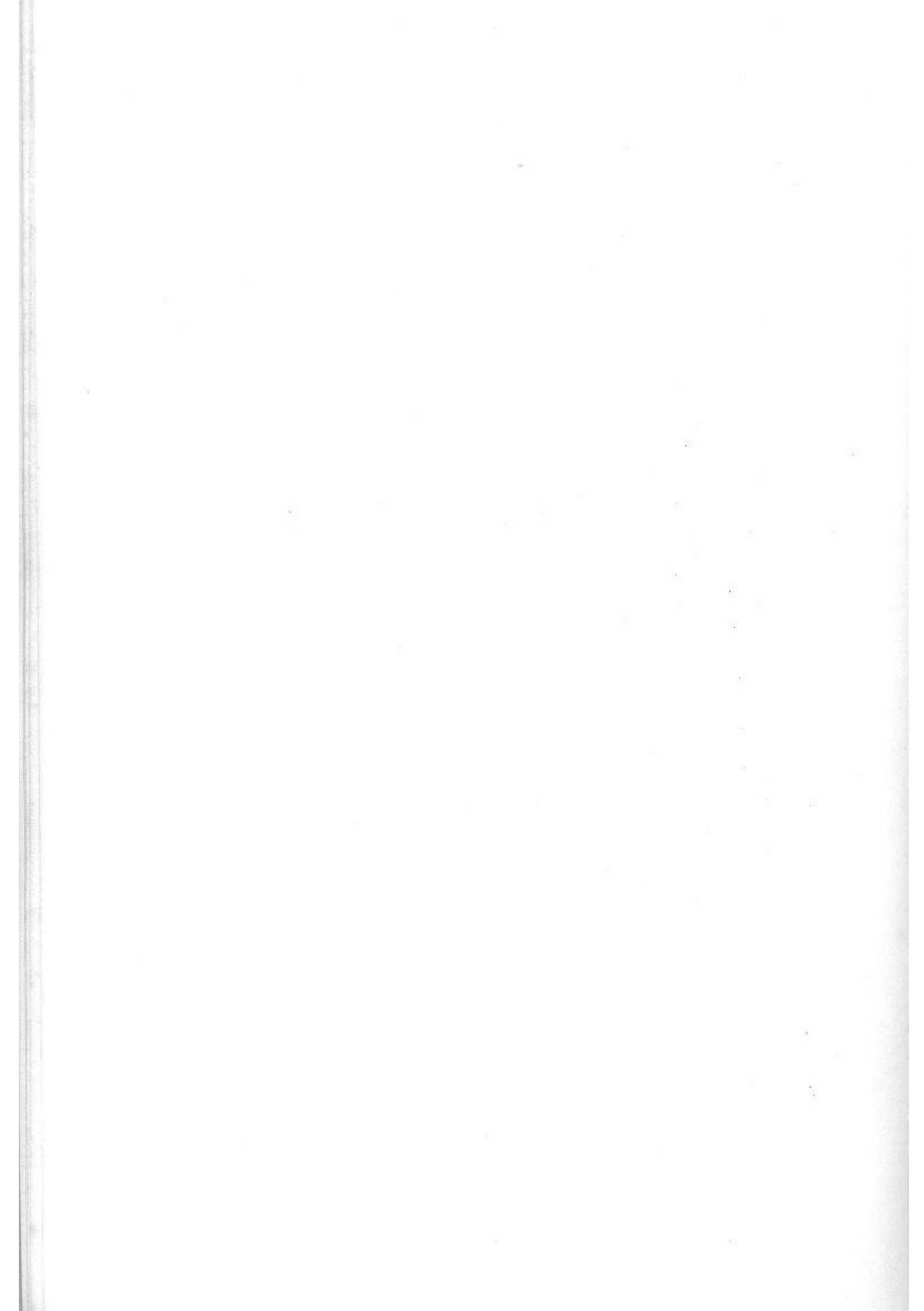
SINTAXE SRL (HL)
SRL (indreg+d)

EXEMPLO SRL (HL)

SRL (IY+8)

FUNÇÃO Esta instrução atua do mesmo modo que a anterior sendo que agora o dado a ser rotacionado encontra-se na posição de memória indicada por HL, IX ou IY.

FLAGS São afetados da mesma forma que nas instruções anteriores.

Com este capítulo encerramos a apresentação das instruções dedicadas ao processamento em geral. No próximo capítulo apresentaremos as instruções que realizam a comunicação do Z80 com o ambiente externo, como o processador de vídeo, o gerador de sons e a interface programável de periféricos, entre outros.

CAPÍTULO 12

AS INSTRUÇÕES DE ENTRADA E SAÍDA DE DADOS

Como já afirmamos anteriormente, o Z80 se comunica com os demais componentes do MSX através de instruções especiais, como as instruções IN e OUT. Neste capítulo iremos apresentar o grupo de instruções responsáveis pela comunicação do Z80 com diversos componentes como: gravador K-7, disk driver e impressora. Sempre que o Z80 necessita de saber qualquer coisa de algum periférico, devemos realizar uma instrução IN e se quisermos que o Z80 envie alguma informação para um determinado periférico devemos usar uma instrução OUT. Segue-se a lista das instruções disponíveis:

SINTAXE IN A,(porta)

EXEMPLO IN A,(A8H)

FUNÇÃO Esta instrução lê um byte da porta especificada. Cabe agora especificar que o Z80 pode "endereçar" 2 ^ 8 (256) portas, que são os meios utilizados pelo Z80 para se comunicar com os demais componentes do MSX.

FLAGS Não são afetados por esta instrução.

SINTAXE IN reg,(C)

EXEMPLO IN B,(C)

FUNÇÃO Esta instrução lê um byte da porta indicada pelo registro C e coloca o valor lido no registro especificado reg.

FLAGS Os flags AC, Z e S passam a refletir o resultado da operação, o flag N é ressetado, o flag P/O é colocado em P e o flag C não é alterado.

SINTAXE IND

EXEMPLO IND

FUNÇÃO Esta instrução lê um byte da porta indicada pelo registro C e o coloca no endereço de memória indicado por HL. Logo após esta instrução decrementa HL e o registro B.

FLAGS O flag N é colocado em um, o flag Z representa o resultado da operação, o flag do CARRY não é alterado e os demais flags assumem valores sem nexo aparente.

SINTAXE INDR

EXEMPLO INDR

FUNÇÃO Esta instrução realiza o mesmo que a operação anterior só que repete a leitura até que o registro B seja igual a zero. O registro HL que indica as posições de memória,

é decrementado cada vez que o registro B é decrementado. Note que a transferência para a memória se dá, portanto, dos endereços mais altos para os mais baixos.

FLAGS São afetados da mesma maneira que pela instrução anterior.

SINTAXE INI

EXEMPLO INI

FUNÇÃO Esta instrução lê um byte da porta especificada pelo registro C e o coloca na posição de memória indicada por HL. Logo após esta instrução decrementa B e incrementa HL.

FLAGS São afetados da mesma forma que nas duas instruções anteriores.

SINTAXE INIR

EXEMPLO INIR

FUNÇÃO Esta instrução realiza quase a mesma função que a anterior, sendo que agora a transferência de dados para a memória é de tantos bytes quantos forem indicados pelo registro B. Como se pode observar, a transferência, desta vez, é para endereços crescentes de memória.

FLAGS São afetados da mesma forma que nas últimas instruções analisadas.

SINTAXE OUT (porta),A

EXEMPLO OUT (A8H),A

FUNÇÃO Esta instrução tem como função enviar um dado de 8 bits contido no acumulador para a porta especificada, que por sua vez realiza a comunicação entre o Z80 e um determinado periférico.

FLAGS Não são alterados por esta instrução.

SINTAXE OUT (C),reg

EXEMPLO OUT (C),D

FUNÇÃO Esta instrução envia para a porta especificada pelo registro C o dado contido no registro de 8 bits reg.

FLAGS Não são alterados por esta instrução.

SINTAXE OUTD

EXEMPLO OUTD

FUNÇÃO Esta instrução envia o dado presente no endereço de memória apontado por HL, para a porta especificada pelo registro C. Logo após decrementa HL e B.

FLAGS O flag N é setado, o flag Z é alterado pelo decremento do registro B, o flag C não é alterado e os demais assumem estados desconhecidos.

SINTAXE OTDR

EXEMPLO OTDR

FUNÇÃO Esta instrução envia o dado presente na memória endereçada por HL, para a porta especificada pelo registro C e a seguir decrementa B e HL e repete o processo até que B seja igual a zero.

FLAGS São afetados da mesma forma que na instrução anterior.

SINTAXE OUTI

EXEMPLO OUTI

FUNÇÃO Esta instrução envia o dado presente na memória endereçada por HL, para a porta especificada pelo registro C e logo após, decrementa o registro B e incrementa o registro HL.

FLAGS São alterados da mesma forma que pelas duas instruções anteriores.

SINTAXE OTIR

EXEMPLO OTIR

FUNÇÃO Esta instrução envia o dado presente na memória endereçada por HL, para a porta especificaca pelo registro C e logo após, decrementa o registro B e incrementa HL, repetindo o processo até que B seja igual a zero.

FLAGS São alterados da mesma forma que nas três últimas instruções analisadas.

O próximo capítulo trata de instruções que devido ao objetivo introdutório deste livro não foram incluídas em nenhum dos grandes grupos de instruções.

CAPÍTULO 13

AS INSTRUÇÕES DE RESTART E INTERRUPÇÃO

As instruções de **RESTART** podem ser entendidas como instruções do tipo CALL de um só byte. Ao todo o Z80 possui oito instruções do tipo RESTART que provocam chamadas de sub-rotinas nos endereços 00H, 08H, 10H, 13H, 20H, 28H, 30H, e 38H. Como você pode observar, os endereços de chamada encontram-se nos primeiros 256 bytes de memória, assim sendo quando operamos o BASIC MSX, estas rotinas se encontram na memória ROM. A abreviação usada pelos mnemônicos desta instrução é a seguinte:

RST 00H

Neste caso a instrução chamaria a rotina de partida do micro, o que provocaria um RESET no sistema.

O outro grupo de instruções que não foi mencionado é o das interrupções. No livro de programação avançada em assembly para MSX a ser lançado após este, iremos descrever este grupo com mais detalhes.

Por ora basta saber que quando ocorre uma interrupção o Z80 pára o seu processamento normal para atender o dispositivo que lhe enviou a interrupção. No caso do MSX temos uma interrupção para atualizar o relógio interno do MSX. Esta interrupção é feita pelo VDP que no caso dos computadores MSX nacionais se repete 60 vezes por segundo. O que ocorre é que o VDP interrompe o trabalho normal do Z80 para que ele atualize o valor do relógio. Geralmente ao executar uma interrupção, os registros são salvos e a sub-rotina da interrupção é executada até o fim, quando então os registros são recuperados e o Z80 volta ao processamento normal através da instrução RETI (retorno da interrupção). Existem ainda outras duas instruções de interesse: a DI e a EI. A primeira desabilita as interrupções mascaráveis (como a gerada pelo VDP) o que aumenta um pouco a velocidade de processamento pois o Z80 não é interrompido 60 vezes por segundo; já a segunda (EI) faz o contrário: habilita as interrupções. Um cuidado todo especial deve ser tomado quando se usar a instrução DI, pois se voltarmos ao BASIC sem o correspondente EI, perderemos a leitura de teclado que é realizada por uma interrupção.

Além das instruções vistas acima, existem outras como a instrução HALT que simplesmente pára o processamento normal até que uma interrupção seja feita. Existem também instruções para manipular o CARRY, como a instrução SCF, que coloca o CARRY em um e a instrução CCF que complementa o estado atual do CARRY colocando-o em zero se ele for um e vice-versa, alterando para isso o flag N que passa a zero em ambas as instruções.

Existem algumas instruções desprezadas por nós, mas que serão vistas em capítulos posteriores. A partir do próximo capítulo passaremos a aplicar os conceitos vistos até agora na programação em linguagem de máquina do MSX.

CAPÍTULO 14

A INTERAÇÃO ENTRE O BASIC E A LINGUAGEM DE MÁQUINA

Para informarmos a CPU para que deixe de executar o interpretador BASIC e passe a executar alguma rotina em linguagem de máquina definida pelo usuário, existe a instrução **DEFUSR.** No manual de BASIC que acompanha o seu micro-computador, você poderá obter algumas informações sobre esta instrução. O interpretador BASIC do MSX nos permite usar até 10 instruções DEFUSR diferentes, numeradas de 0 a 9 segundo a sintaxe a seguir:

```
DEFUSR0=&HA000
DEFUSR1=&HA010
        .
        .
        .
DEFUSR9=&HB000
```

Note que o valor após o sinal de igual refere-se ao endereço de memória a partir do qual se econtra a sua rotina em linguagem de máquina.

Com o comando DEFUSR nós apenas informamos o endereço da nossa sub-rotina ao interpretador. Para executá-la dispomos da instrução USR, que deve ser usada de acordo com a sintaxe a seguir:

USR0(0)
USR1(0)
.
.
.
USR9(0)

O valor entre parênteses recebe o nome de argumento e é através dele que podemos passar valores do BASIC para a nossa rotina em linguagem de máquina. O argumento pode ser tanto uma constante, como 0, 3.1415926 ou "MSX", como uma variável como A%, A$, A! ou A#. Levando isto em conta as instruções seguintes estão sintaticamente corretas:

USR(A)
USR("MSX")
USR(3.1415)
USR(A%)

Como você deve ter notado não especificamos o índice (0 a 9) da instrução USR. Quando isto acontece o interpretador BASIC assume que estamos trabalhando com a USR0. Outra particularidade que merece ser destacada, é o fato de que para o interpretador BASIC se desviar para a sub-rotina definida pela instrução DEFUSR, devemos utilizar sentenças do tipo:

PRINT USR0(1)
ou
LET A=USR0(1)

O parâmetro de uma instrução USR é muito importante, como vere-

mos a seguir, para otimizar ainda mais a interação BASIC-LINGUAGEM DE MÁQUINA.

OS PARÂMETROS DA INSTRUÇÃO USR

Como já afirmamos, o parâmetro é útil por que com ele podemos passar valores do BASIC para o nosso programa em assembly de uma maneira racional e rápida. Para que isso aconteça o interpretador BASIC reserva uma área de memória RAM para guardar todas as informações necessárias sobre o argumento. Esta área se chama ZONA DE PARÂMETRO e guarda as informações pertinentes na seguinte ordem:

1º. Valor real do parâmetro representado da maneira mais conveniente possível, conforme seja inteiro, real ou string;

2º. A informação do endereço do parâmetro na memória.

A zona de parâmetro ocupa duas áreas distintas da zona de trabalho do sistema; uma área é de 1 byte apenas e a outra é de 8 bytes. A primeira área de um byte de comprimento indica o tipo de argumento (inteiro, simples precisão, dupla precisão ou string) e se encontra no endereço F663H (63075D), já a segunda área, de 8 bytes de comprimento, armazena o valor propriamente dito do argumento. Esta área estende-se dos endereços F7F6H (63478D) a F7FDH (63485D). Para não fugir do caráter introdutório deste livro nos deteremos apenas nos argumentos inteiros e alfanuméricos.

OS ARGUMENTOS INTEIROS

Os números inteiros são todos aqueles compreendidos na faixa de -32768D a 32767D sem parte fracionária evidentemente. Em BASIC

temos a instrução DEFINT para designar as variáveis que deverão receber valores inteiros. A designação do tipo LET A%=3 também cria em BASIC uma variável inteira. As vantagens de se trabalhar com variáveis inteiras são várias, sendo as principais o fato de se gastar menos memória, pois gasta-se somente dois bytes para armazenar o valor de um inteiro, e também, o aumento da velocidade de processamento pois o sistema só irá manipular 2 bytes e as operações de soma e subtração podem ser feitas usando-se somente as potencialidades do Z80.

Vejamos agora como armazenar um valor inteiro na memória usando somente dois bytes. Seja, por exemplo, o número 40F0H: se quisermos armazená-lo a partir da posição F000H, basta dar, através do próprio BASIC, dois pokes, quais sejam:

POKE &HF000,&HF0
POKE &HF001,&H40

Como você percebeu, bastou-nos dividir o número em duas partes de 2 algarismos cada e armazenar a última parte na primeira posição de memória e a primeira parte na segunda posição de memória. Por convenção, chama-se a segunda parte, no caso do nosso exemplo o valor F0H, de LSB e a primeira de MSB. O termo LSB é uma abreviação da expressão inglesa (Least Significant Byte) – byte de valor menos significativo e o termo MSB (Most Significant Byte), o byte de valor mais significativo. No caso do exemplo temos LSB=F0H e MSB=40H. Suponha que desejássemos trabalhar na base decimal ao invés da hexadecimal, no caso do exemplo anterior teríamos 40F0H que é igual a 16624D. O LSB seria 16624-INT(16624/256) * 256, donde tiramos LSB=240D=F0H, e o MSB seria INT(16624/256) que dá 64D ou 40H. Como você deve ter observado é muito mais simples trabalhar na base hexadecimal do que na base decimal no caso da aritmética nos computadores. Embora o método usado pelo Z80 para armazenar números inteiros lhe pareça complicado, não se assuste, pois logo você se acostumará a lidar com os números armazenados ao contrário.

Agora que já sabemos como são armazenados os valores inteiros, resta-nos informar que, quando o interpretador BASIC percebe que o argumento é inteiro, coloca na posição F663H o valor 2, indicando que o valor contido na zona de parâmetro é inteiro e como tal deve ser tratado, e armazena o valor do argumento propriamente dito nas posições F7F8H e F7F9H no LSB e MSB, como vimos anteriormente.

OS ARGUMENTOS ALFANUMÉRICOS

Ao passarmos um argumento alfanumérico de um programa em BASIC para uma rotina em linguagem de máquina, a posição de memória F663H da zona de parâmetro, passará a conter o valor 3, indicando ao sistema que o argumento é alfanumérico. Como sabemos, uma variável alfanumérica pode ter um tamanho máximo de 255 caracteres, surge então a pergunta: como armazenar a cadeia de caracteres se a zona de parâmetros só tem capacidade para 8 caracteres? A resposta é simples, pois o sistema coloca nas posições F7F8H e F7F9H o endereço a partir do qual encontramos o tamanho da string bem como o seu endereço dentro do texto do programa em BASIC que ativou a rotina em linguagem de máquina. A forma de armazenamento do endereço da cadeia de caracteres obedece a regra LSB e MSB, vistas anteriormente.

Até agora vimos como passar valores do BASIC para rotinas em linguagem de máquina mas não o contrário. Acontece que se tivermos a seguinte sentença:

PRINT USR0(78)

ao terminar a execução da rotina em linguagem de máquina o sistema imprimirá na tela o valor 78. Como você pode observar, isto não tem uma grande utilidade.

COMO PASSAR PARÂMETROS DE VOLTA AO BASIC

Já vimos como passar argumentos do BASIC para rotinas em linguagem de máquina e analisamos o que acontece no retorno dessas rotinas ao BASIC. Embora no tópico anterior você possa ter ficado com a impressão que é muito difícil passar parâmetros das rotinas em linguagem de máquina para o BASIC, eu lhe garanto que é bem simples. O segredo se resume em realizar as operações inversas às realizadas para passar os argumentos do BASIC para as rotinas.

Suponha que haja a necessidade de passar um valor inteiro obtido na rotina em linguagem de máquina, para o BASIC. A primeira coisa a fazer seria colocar na posição F663H o valor 2 para indicar ao sistema que o argumento a passar para o BASIC é inteiro, e a segunda medida, seria colocar nas posições F7F8H e F7F9H o valor na forma LSB e MSB. Se o valor a ser passado estivesse no par de registros HL, o trecho final da sub-rotina poderia ser:

```
LD      A,02H
LD      (0F663H),A
LD      (0F7F8H),HL
RET
```

O caso de se passar parâmetros alfanuméricos das rotinas em linguagem de máquina para o BASIC é um pouco mais complicado e será deixado para um capítulo posterior.

Neste capítulo aprendemos a passar parâmetros do BASIC para rotinas em linguagem de máquina e a passar parâmetros inteiros de rotinas em linguagem de máquina para o BASIC. Nos próximos capítulos usaremos exaustivamente os conceitos vistos até agora, pois devido ao caráter introdutório deste livro, muitos dos nossos exemplos apresentarão interações com o BASIC e a linguagem de máquina.

CAPÍTULO 15

A INTERFACE PROGRAMÁVEL DE PERIFÉRICOS (PPI)

Este capítulo será dividido em duas partes. Na primeira apresentaremos todas as potencialidades desse processador, e na segunda apresentaremos exemplos práticos de como programá-lo em linguagem de máquina.

O PPI apresenta quatro portas através das quais informa ao sistema o status atual da máquina. Através destas quatro portas o Z80 lê o teclado, chaveia os bancos de memória disponíveis, liga e desliga o motor do gravador cassete, liga e desliga a lâmpada de caps lock e controla o click do teclado manipulando o PSG. Como você pode observar, este chip está diretamente ligado com uma grande parte das rotinas de entrada e saída de dados. Passemos então à descrição das portas.

PORTA A (ENTRADA E SAÍDA PELA PORTA A8H)

No título colocamos a sentença "entrada e saída", que significa que tanto podemos ler dados como gravá-los, utilizando para isso as instruções IN e OUT respectivamente. Como sabemos, o Z80 é um microprocessador de 8 bits e como tal só pode endereçar um máximo de 64Kb de memória, seja ela do tipo RAM ou do tipo ROM, ou ainda, composta por uma combinação de ambas. Para contornar esta limitação o MSX apresenta a capacidade de chaveamento de slots, que permite em tese um endereçamento total de 1 megabyte de memória alocados em 16 bancos de 64Kb. Assim sendo, o MSX pode chegar a um máximo de 16 slots, sendo quatro principais e quatro secundários para cada um dos principais. Nos MSX nacionais só percebemos a presença de 2 slots, que expandidos poderiam se transformar em 8, permitindo assim o endereçamento total de 512 Kb. Os outros dois slots não visíveis estão ocupados pela ROM do MSX BASIC (slot 0) e pelos 64Kb de memória RAM (slot 2 nos micros de Gradiente e slot 3 nos micros da Sharp). Com este mecanismo de chaveamento do slots poderíamos possuir um compilador FORTRAN atuando num slot, um compilador BASIC atuando num outro e assim por diante mostrando o quanto é poderoso este mecanismo.

A porta A8H informa ao Z80 onde estão posicionados, pelos diversos slots disponíveis, os quatros bancos de 16Kb de memória que irão constituir os 64Kb máximos permitidos. A cada um destes bancos dá-se o nome de página. Assim temos a página zero que contém os primeiros 16Kb de memória, a página um os segundos 16Kb, a página dois o terceiro banco de 16Kb e a página três os últimos 16Kb disponíveis. Como só são possíveis 4 slots (0 a 3) só necessitamos de 2 bits para designá-los e é desta forma que o PPI informa o Z80 como você poderá perceber na figura a seguir:

bits 7 6	5 4	3 2	1 0
Pag. 3 Slot Prin.	Pag. 2 Slot Prin.	Pag. 1 Slot Prin.	Pag. 0 Slot Prin.

Ao ligar o seu micro, a porta A8H apresentará o valor 10100000B se o seu micro for da Gradiente e 11110000B se for da Sharp. Analisando estes valores você pode observar que as páginas 0 e 1 apontam para o slot 0 onde se encontra a ROM de 32Kb do MSX BASIC, já as páginas 2 e 3 apontam para o slot de memória RAM, apresentando assim uma configuração de 32Kb de ROM e 32Kb de RAM. Se você quiser obter a leitura desta porta, digite a seguinte linha em BASIC:

PRINT RIGHT$("00000000"+BIN$(INP(&HA8)),8);"B"

Não é demais lembrá-lo que as operações envolvendo esta porta devem ser cercadas de todos os cuidados possíveis, pois se erramos na configuração podemos provocar um crash no sistema que, embora não cause avarias à máquina, poderá provocar a perda dos dados gravados na memória.

PORTA B (ENTRADA E SAÍDA PELA PORTA A9H)

Esta porta é a responsável pela leitura do teclado, que é feita inteiramente por software. Por intermédio desta porta podemos ler os 8 bits correspondentes às 8 colunas integrantes de uma linha especificada. A leitura por software considera o teclado do MSX como sendo composto por uma matriz de 8 colunas por 11 linhas, sendo que os micros nacionais utilizam 9 linhas, como é o caso do micro da Sharp, ou 10 linhas como é o caso do micro da Gradiente. A seguir temos a matriz do teclado do padrão MSX:

REGISTRO B / REGISTRO C (4 bits menos significativos)

REGISTRO C	7 (MSB)	6	5	4	3	2	1	0 (LSB)
0000	7	6	5	4	3	2	1	0
0001	;	[	@	¥	∧	–	9	8
0010	B	A	–	/	.	,	]	:
0011	J	I	H	G	F	E	D	C
0100	R	Q	P	O	N	M	L	K
0101	Z	Y	X	W	V	U	T	S
0110	F3	F2	F1	かな	CAPS	GRAPH	CTRL	SHIFT
0111	RETURN	SELECT	BS	STOP	TAB	ESC	F5	F4
1000	→	↓	↑	←	DEL	INS	HOME CLS	SPACE

PORTA C (ENTRADA E SAÍDA PELA PORTA AAH)

Esta porta é responsável pelo controle de diversas funções como podemos verificar, examinando a figura seguinte:

7	6	5	4	3	2	1	0
Click do teclado	Luz de C.Lock	Saída para o K-7	Motor do K-7	Selecionador das linhas do teclado			

Bits 0 a 3 São os responsáveis pela seleção da linha de teclado onde estão as teclas que desejamos verificar se estão presssionadas ou não. A seleção das linhas se faz dentro dos valores 0 a 10 representando as 11 linhas possíveis.

Bit 4 Este bit chaveia o liga/desliga do motor do gravador K-7. Se for igual a zero liga o motor e se for igual a 1 desliga.

Bit 5 Este bit é o que efetivamente envia os dados para o gravador K-7. Antes de chegar ao gravador o sinal deste bit é atenuado e filtrado pelo circuito de FSK. Toda a geração das freqüências para gravação de dados é feita por Soft.

Bit 6 Este bit determina o estado da lâmpada de caps lock, se for 0 a lâmpada se acende e se for 1 se desliga.

Bit 7 Este bit ativa ou desativa o click do teclado, conforme seja 0 ou 1 respectivamente. O sinal enviado por este bit é analisado e mixado pelo PSG.

PORTA DE MODO DE OPERAÇÃO (ENTRADA E SAÍDA PELA PORTA ABH)

Esta porta é usada para indicar ao próprio PPI o seu modo de operação, por este motivo devemos ter muito cuidado quando alterarmos o seu valor, pois o MSX foi projetado para trabalhar de uma forma específica. Se por exemplo especificarmos a porta B pára escrita (normalmente é de leitura) e a porta C para escrita (como é normalmente) poderemos causar a queima deste circuito, portanto muito cuidado ao manipular esta porta.

A figura a seguir esclarece a função de cada um dos bits que compõem os dados desta porta:

7	6	5	4	3	2	1	0
1	Modo das portas A e C		Dir. da porta A	Dir. da porta C	Modo das port. B e C	Dir. da porta B	Dir. da porta C

Bit 7 Este bit deve ser mantido em 1 para permitir alterações no modo de operação do PPI. Caso contrário passamos a manipular somente a porta C.

Bits 6 e 5 Estes bits determinam o modo de operação da porta A e dos quatro bits superiores (4 a 7) da porta C. O valor normal em que não deve ser mudado é 00.

Bit 4 Este bit indica a direção da porta A. O valor 0 indica que a porta A está para escrita (valor normal do MSX), já o valor 1 indica que a porta está para leitura.

Bit 3 Este bit indica a direção dos quatro bits mais altos (4 a 7) da porta C. Se o valor for 0 estão para escrita (valor normal do MSX), se for 1 estão para leitura.

Bit 2 Este bit indica o modo de operação da porta B e dos quatro bits mais baixos da porta C. O valor normal é 0.

Bit 1 Este bit indica a direção da porta B. O valor 0 indica que a porta B está para escrita, já o 1 indica que está para leitura, que é o modo normal no MSX.

Bit 0 Este bit indica a direção dos quatro bits mais baixos (0 a 3) da porta C. O valor 0 indica que estão para escrita

(valor normal do MSX), já o valor 1, que estão para leitura.

Ao modificarmos o bit 7 desta porta para o valor 0, o PPI pode ser usado para setar ou ressetar diretamente qualquer bit da porta C. A figura a seguir ilustra esta alornativa:

7	6	5	4	3	2	1	0
0	**Não são usados**			**número do bit**			**Set Reset**

Pela figura acima podemos verificar que se, quisermos modificar algum bit diretamente basta-nos informar a porta ABH o número do bit a ser modificado e depois colocar o seu valor no bit 0 da referida porta. Para exemplificar, ligue o micro e o gravador com os pinos do remoto ligado e digite as seguintes linhas em BASIC:

OUT &HAB,&B00001000

OUT &HAB,&B00001001

A primeira linha irá acionar o motor do gravador K-7 e a segunda irá pará-lo. Da mesma forma se você possui o micro da Sharp digite:

OUT &HAB,&B00001100

OUT &HAB,&B00001101

A primeira linha irá acender a lâmpada de caps lock e a segunda irá desligá-la.

Após esta apresentação das potencialidades passaremos à fase de programação em linguagem de máquina do PPI.

EXEMPLOS PRÁTICOS DE PROGRAMAÇÃO DO PPI

No primeiro exemplo iremos provocar variações muito rápidas no valor do bit 7 da porta C para produzir sons diversos. O montador assembler usado por nós é o SIMPLE ASM da CORAL, vendido por quase todas as empresas que negociam com a venda de programas. Se você possuir um montador destes, digite a listagem abaixo e teste, caso contrário aconselho-o a comprar um.

1º. Exemplo: Emissão de sons diversos

```
1000          ORG     0E000H
1010          DI
1020          LD      HL,(0F7F8H)
1030  LOOP1:  LD      A,00001111B
1040          OUT     (0ABH),A
1050          LD      B,0FFH
1060  LOOP2:  DJNZ    LOOP2
1070          LD      A,00001110B
1080          OUT     (0ABH),A
1090          LD      B,0FFH
1100  LOOP3:  DJNZ    LOOP3
1110          DEC     HL
1120          LD      A,H
1130          OR      L
1140          JR      NZ,LOOP1
1150          EI
1160          RET
```

Na linha 1010 desabilitamos as interrupções para tornar o processamento mais uniforme. Na linha 1020 pegamos a duração do som como parâmetro da função USR. Nas linhas 1030 e 1040 emitimos um sinal para o alto-falante. Nas linhas 1050 e 1060 provocamos uma pequena pausa para modular a freqüência do som emitido. Da linha 1070 à li-

nha 1100 repetimos o procedimento para a ausência de sinal, e da linha 1110 a 1140 testamos o fim da emissão do som. Nas linhas 1150 e 1160 preparamos a volta ao BASIC. Após a compilação do programa acima digite em BASIC:

DEFUSR=&HE000: A%=1000: PRINT USR(A%)

Com este comando você irá ouvir um som de freqüência constante. Se quiser outras freqüências experimente alterar o valor do registro B nas linhas 1050 e/ou 1090.

Como você pode notar, o programa acima nada mais faz que gerar uma onda quadrada e a freqüência com que ela oscila é que produz o som.

2º. Exemplo: Emissão de sons para o gravador K-7

```
1000          ORG     0E000H
1010          DI
1020          LD      HL,(0F7F8H)
1030          LD      A,00001000B
1040          OUT     (0ABH),A
1050  LOOP1:  LD      A,00001010B
1060          OUT     (0ABH),A
1070          LD      B,0FFH
1080  LOOP2:  DJNZ    LOOP2
1090          LD      A,00001011B
1100          OUT     (0ABH),A
1110          LD      B,0FFH
1120  LOOP3:  DJNZ    LOOP3
1130          DEC     HL
1140          LD      A,H
1150          OR      L
1160          JR      NZ,LOOP1
1170          LD      A,00001001B
```

```
1180        OUT         (0ABH),A
1190        EI
1200        RET
```

2º. Exemplo: Emissão de sons para o gravador K-7

O funcionamento deste programa é semelhante ao anterior, sendo que agora a oscilação é sobre o bit responsável pela gravação de dados na fita K-7. Para vê-lo funcionar, compile o programa, ligue os cabos ao gravador e prepare-o para gravação, logo a seguir digite em BASIC:

DEFUSR=&HE000: A%=1000: PRINT USR(A%)

3º. Exemplo: Ler a linha dois da matriz do teclado

```
1000        ORG         0E000H
1010        DI
1020        IN          A,(0AAH)
1030        AND         11110000B
1040        OR          00000010B
1050        OUT         (0AAH),A
1060        IN          A,(0A9H)
1070        LD          (0F7F8H),A
1080        XOR         A
1090        LD          (0F7F9H),A
1100        EI
1110        RET
```

Da linha 1020 a 1050 preparamos a linha dois do teclado para leitura, na linha 1060 lemos a linha selecionada e da linha 1070 a 1090 preparamos a transferência do valor lido para o BASIC, e nas linhas 1100 e 1110 preparamos a volta ao BASIC. Para testar o programa, após a compilação, digite o programa BASIC seguinte:

```
10 CLS:KEYOFF:DEFUSR=&HE000
```

```
20 A=USR(0):LOCATE 15,10
30 PRINT RIGHT$("00000000"+BIN$(A),8)
40 GOTO 20
```

O programa acima testa a linha que compreende as teclas B e A entre outras, assim sendo rode o programa e pressione uma tecla qualquer que faça parte desta linha de teclado. Como você poderá observar, o bit correspondente à tecla pressionada apresentará o valor zero, e as demais não pressionadas o valor 1. Experimente pressionar várias teclas e observe o que acontece. Esta leitura poderá ser usada para ler várias teclas pressionadas simultaneamente, como ocorre constantemente nos jogos de ação.

Com este capítulo encerramos a apresentação e programação do PPI. Nos próximos capítulos nos dedicaremos à apresentação e programação dos processadores de vídeo e de som.

CAPÍTULO 16

O OPERADOR PROGRAMÁVEL DE SOM (PSG)

No capítulo anterior, mostramos uma pequena rotina em linguagem de máquina, capaz de produzir sons de diversas freqüências. Acontece porém, que esta não é a maneira mais inteligente de se produzir sons num MSX, pois toda vez que necessitarmos de gerar algum som interrompemos o trabalho da CPU aumentando assim o tempo de processamento do programa propriamente dito. Este sistema é empregado em micros de tecnologia não tão avançada como o ZX SPECTRUM, o TRS-80 e o APPLE II. No MSX temos um chip cuja principal função é produzir sons. A vantagem deste dispositivo é a de produzir até 3 sons distintos ao mesmo tempo com uma intervenção mínima da CPU, que assim é liberada mais cedo para outras tarefas.

O gerador de sons do MSX é um dispositivo com capacidade para três canais, o que significa, que habilita o MSX a produzir três sons distintos, cada qual com tonalidade e volume distintos dos outros dois se assim quisermos. Outra característica interessante é a de permitir que cada um destes canais gere ruídos, permitindo assim efeitos especiais como explosões, tiros, etc.

A forma empregada para se programar o PSG é semelhante a empregada para programar o PPI. Assim senco, podemos descrever o PSG como um processador com duas portas de dados de 8 bits, chamadas A e B, através das quais se faz o interfaceamento entre ele e os joysticks e a leitura de dados do gravador K-7. O PSG aparece para o Z80 como três portas chamadas de: porta de endereço, porta de emissão de dados e porta de leitura de dados. A partir de agora passamos a descrever as funções de cada uma destas portas.

PORTA DE ENDEREÇO (ENTRADA E SAÍDA PELA PORTA A0H)

Esta porta informa ao PSG o número do registro que queremos modificar ou ler. O PSG possui 16 (0 a 15) registros com diversas funções, que serão apresentadas mais adiante. Uma vez selecionado o registro, o acesso a ele poderá ser feito continuamente pelas portas de leitura e escrita.

PORTA DE ESCRITA (ENTRADA E SAÍDA PELA PORTA A1H)

Esta porta é usada para gravar um valor no registro previamente selecionado pela porta A0H.

PORTA DE LEITURA (ENTRADA E SAÍDA PELA PORTA A2H)

Esta porta é usada para ler o estado atual de um registro selecionado previamente pela porta A0H.

OS REGISTROS DO PSG E SUAS FUNÇÕES

Registros 0 e 1

Estes dois registros são usados para definir a freqüência do tom gerado pelo canal A. O número de bits que compõem a freqüência do tom é 12, sendo 8 no registro 0 e 4 no registro 1. Os quatro bits do registro 1 compõem o valor mais significativo (MSB) da freqüência, e por este motivo é chamado de registro de CONTROLE DOS GRAVES do tom. O registro 0 contém os 8 bits menos significativos (LSB) da freqüência, e por isso, é chamado de registro de CONTROLE DOS AGUDOS do tom. Como o valor da freqüência é composto por 12 bits, podemos ter 4096 freqüências diferentes (1 a 4095). O PSG divide uma freqüência padrão de 1,7897725MHz por 16, para obter a maior freqüência possível, que é 111.861Hz. A freqüência externa vai então de 111.861Hz (dividida por 1) a 27,3Hz (dividida por 4095).

<table>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td colspan="8">Freqüência do canal A
Valor LSB</td><td>R0</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td colspan="4">Freqüência do canal A
Valor MSB</td><td>R1</td></tr>
</table>

Registros 2 e 3

Estes dois registros controlam a freqüência do canal B, da mesma forma que a vista nos registros 0 e 1.

Registros 4 e 5 Estes dois registros controlam a freqüência do canal C da mesma forma que a vista no controle da freqüência do canal A.

Registro 6 Além dos três canais vistos anteriormente, o PSG possui um gerador de ruído simples cuja freqüência é dada pelo valor de cinco bits deste registro. Assim sendo temos 32 freqüências diferentes (1 a 31) que são manipuladas pelo PSG dividindo a freqüência fundamental de 111.861Hz pelo valor contido neste registro. Assim sendo a faixa da freqüência de ruído vai de 3.608Hz a 111.861Hz.

7	6	5	4	3	2	1	0
X	X	X	Freqüência do ruído				

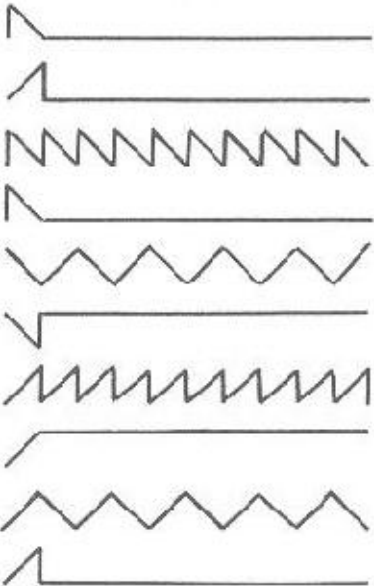
Registro 7 Este registro habilita ou desabilita o gerador de sons e o gerador de ruídos de cada um dos três canais. O valor 0 habilita, e o valor 1 desabilita. Os bits 7 e 6 deste registro especificam a direção das portas B e A respectivamente, que como mencionado anteriormente fazem o interfaceamento entre o PSG e os joysticks e a entrada de dados pelo gravador K-7. O valor 0 especifica que a porta determinada está pronta a receber dados, e o valor 1, que está pronta a enviar dados.

7	6	5	4	3	2	1	0
Dir. da Porta B	Dir. da Porta A	Ruído do Canal C	Ruído do Canal B	Ruído do Canal A	Som do Canal C	Som do Canal B	Som do Canal A

Registro 8 Este registro determina a amplitude do som gerado pelo canal através do valor dos 4 bits menos significativos. O bit 4 seleciona entre amplitude fixa ou modulada. Se o valor for 0 a amplitude será fixa, se 1 a amplitude será modulada na saída pelo gerador de envelope. Quando nos referimos a amplitude estamos nos referindo da mesma forma ao volume.

7	6	5	4	3	2	1	0
X	X	X	Modo	Amplitude do canal A			

Registro 9 A função deste registro é a de controlar a amplitude do canal B da mesma forma que o registro 8 controla a do canal A.

Registro 10 A função deste registro é a de controlar a amplitude do canal C da mesma forma que o registro 8 controla a do canal A.

Registros 11 e 12 Estes dois registros determinam a freqüência do gerador de envelope usado na modulação da amplitude. Como o valor é composto por 16 bits, podemos gerar envelopes de 65536 (1 a 65535) freqüências diferentes. O valor menos significativo (LSB) encontra-se no registro 11 e o mais significativo (MSB) no registro 12. A freqüência padrão neste caso é de 6.991Kz, logo a faixa de freqüência do envelope vai de 6.991Hz (divisão por 1) a 0,11Hz (divisão por 65535).

7	6	5	4	3	2	1	0	
Freqüência do envelope (Valor LSB)								R11
Freqüência do envelope (Valor MSB)								R12

Registro 13 Este registro determina a forma do envelope dentre 10 formas disponíveis, através do valor representado pelos quatro bits menos significativos do registro.

7	6	5	4	3	2	1	0
X	X	X	X	Forma do envelope			

ENVELOPE DE MODULAÇÃO

3	2	1	0
0	0	x	x
0	1	x	x
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Registro 14 Este registro é usado para ler a porta A do PSG que como afirmamos anteriormente faz o interfa-

ceamento dos joysticks e do gravador K-7 com o PSG. Os cinco bits inferiores são usados para ler o estado do joystick selecionado, onde o valor 0 indica que o disparo (ou uma direção específica) está pressionada e o valor 1 que não está pressionado. O bit 6 não é usado nos MSX ocidentais e o bit 7 é usado para ler o sinal que chega pelo EAR do gravador K-7.

7	6	5	4	3	2	1	0
Leit. do K-7	Modo do Tec.	Dis. 1 do Joy.	Dis. 2 do Joy.	Dir. do Joy.	Esq. do Joy.	Tras. do Joy.	Fren. do Joy.

Registro 15

Este registro é usado para enviar dados do PSG para a porta B. Os quatro bits menos significativos são conectados via TTL aos pinos 6 e 7 dos joysticks. Os seus valores são normalmente 1. Os dois bits de pulso (4 e 5) são usados para enviar pulsos a paddles enventualmente ligados nos conectores dos joysticks. O bit 6 é usado para selecionar o joystick, onde o valor 0 indica que foi selecionado o joystick 1 e o valor 1 o joystick 2. Já o bit 7 não é usado nos MSX ocidentais.

7	6	5	4	3	2	1	0
Kana Led	Sel. Joy	Pul. 2	Pul. 1	1	1	1	1

COMO LER E ESCREVER NOS REGISTROS DO PSG

A forma empregada para ler e escrever nos registros do PSG é bem simples, como você poderá comprovar ao examinar o quadro a seguir:

```
Para ler        LD    A,n. do registro
                OUT   (0A0H),A          ;manda-o para o PSG
                IN    A,(0A2H)          ;A=valor do registro

Para escrever   LD    A,n. do registro
                OUT   (0A0H),A
                LD    A,dado            ;A=dado a ser escrito
                OUT   (0A1H),A          ;escreve o dado
```

Depois de todas estas apresentações vamos passar aos exemplos práticos:

1º. Exemplo: Como gerar um som pelo canal A

```
1000          ORG          0E000H
1010          DI
1020          LD           HL,(0F7F8H)
1030          XOR          A
1040          OUT          (0A0H),A
1050          LD           A,(HL)
1060          OUT          (0A1H),A
1070          LD           A,01H
1080          OUT          (0A0H),A
1090          INC          HL
1100          LD           A,(HL)
1110          OUT          (0A1H),A
1120          LD           A,08H
1130          OUT          (0A0H),A
```

```
1140            OUT     (0A1H),A
1150            LD      B,04H
1160  LOOP1:    LD      HL,0FFFFH
1170  LOOP2:    DEC     HL
1180            LD      A,H
1190            OR      L
1200            JR      NZ,LOOP2
1210            DJNZ    LOOP1
1220            LD      A,08H
1230            OUT     (0A0H),A
1240            XOR     A
1250            OUT     (0A1H),A
1260            EI
1270            RET
```

No exemplo acima geramos um som cuja freqüência é passada do BASIC para a rotina como argumento da função USR. Da linha 1040 a linha 1110 preparamos os registros 0 e 1 para escrita com os valores passados pelo argumento da função USR. Da linha 1120 a 1140 especificamos um volume de intensidade 8. Da linha 1160 a 1210 preparamos um loop para que nos dê tempo de ouvir o som gerado, e finalmente da linha 1220 a 1270 colocamos o volume em zero e preparamos a volta ao BASIC. Para testar o programa acima você deverá compilá-lo e depois digitar a seguinte linha em BASIC:

DEFUSR=&HE000:PRINT USR(&H010F)

Como você poderá observar o valor 01H irá determinar o controle dos graves e o valor OFH o controle dos agudos. Tente mudar o argumento da função USR e observe o resultado.

2º. Exemplo: Como gerar ruídos pelo canal A

```
1000            ORG     0E000H
1010            DI
```

```
1020          LD      A,07H
1030          OUT     (0A0H),A
1040          IN      A,(0A2H)
1050          SET     0,A
1060          RES     3,A
1070          LD      C,A
1080          LD      A,07H
1090          OUT     (0A0H),A
1100          LD      A,C
1110          OUT     (0A1H),A
1120          LD      A,06H
1130          OUT     (0A0H),A
1140          LD      A,(0F7F8H)
1150          OUT     (0A1H),A
1160          LD      A,08H
1170          OUT     (0A0H),A
1180          LD      A,(0F7F9H)
1190          OUT     (0A1H),A
1200          LD      B,04H
1210  LOOP1:  LD      HL,0FFFFH
1220  LOOP2:  DEC     HL
1230          LD      A,H
1240          OR      L
1250          JR      NZ,LOOP2
1260          DJNZ    LOOP1
1270          LD      A,08H
1280          OUT     (0A0H),A
1290          XOR     A
1300          OUT     (0A1H),A
1310          EI
1320          RET
```

No exemplo acima, da linha 1020 a 1110 preparamos o canal A para gerar ruído. Da linha 1120 a 1190 pegamos o valor do argumento da função USR para definir a freqüência e o volume do ruído a ser gera-

do. Da linha 1200 a 1260 preparamos um loop que nos permita escutar o ruído gerado, e finalmente, da linha 1270 a 1320 preparamos a volta ao BASIC. Para verificar o funcionamento deste programa, compile-o e depois digite a seguinte sentença em BASIC:

DEFUSR=&HE000:PRINT USR(&H0F08)

Como você poderá observar o valor 08H irá determinar a freqüência do ruído e o valor 0FH irá determinar o volume.

3º. Exemplo: Como ler o Joystick diretamente

```
1000    ORG     0E000H
1010    DI
1020    LD      A,0FH
1030    OUT     (0A0H),A
1040    IN      A,(0A2H)
1050    RES     6,A
1060    LD      C,A
1070    LD      A,0FH
1080    OUT     (0A0H),A
1090    LD      A,C
1100    OUT     (0A1H),A
1110    LD      A,0EH
1120    OUT     (0A0H),A
1130    IN      A,(0A2H)
1140    LD      (0F7F8H),A
1150    XOR     A
1160    LD      (0F7F9H),A
1170    EI
1180    RET
```

No exemplo acima da linha 1020 a 1100 preparamos o joystick 1 para exame. Se quisermos examinar o joystick 2 é só trocarmos a instrução RES por SET na linha 1050. Da linha 1110 a 1160 examinamos o es-

tado atual do joystick selecionado e colocamos os valores lidos no argumento da função USR. Nas linhas 1170 e 1180 preparamos a volta ao BASIC. Para testar este programa você deverá compilá-lo e depois digitar o seguinte programa em BASIC.

```
10 CLS:KEYOFF:DEFUSR=&HE000
20 LOCATE14,10:PRINT RIGHT$("00000000"+BIN$(USR(0) ),8)
30 GOTO 20
```

Ao rodar o programa acima você poderá testar o seu joystick e ao mesmo tempo observar como se processa a leitura dos joysticks em linguagem de máquina.

Com este capítulo terminamos a apresentação e programação do gerador de sons do MSX. No próximo capítulo iremos nos dedicar a apresentação do processador de vídeo, que apresenta uma série de recursos extras, entre eles o de possuir a sua própria memória RAM.

CAPÍTULO 17

O PROCESSADOR DE VÍDEO (VDP)

Este capítulo é dedicado ao estudo do processador auxiliar mais complexo do MSX: o VDP. Ao final deste capítulo, você terá condições de escrever textos, desenhar gráficos e manejar SPRITES em linguagem de máquina.

O VDP tem o nome técnico de TMS9918, e é visto pelo Z80 como duas portas de entrada e saída. A primeira porta (98H) é chamada de porta de dados e, a segunda (99H), de porta de comandos. Embora o VDP tenha a sua própria memória RAM de 16Kb de capacidade de armazenamento, ela não pode ser diretamente acessada pelo Z80, isto significa, que se quisermos ler uma determinada posição de memória do VDP, não podemos empregar instruções do tipo LD A,(end), mas sim, OUT's e IN's como ficará claro mais tarde.

A utilidade de VRAM está associada ao fato de que todas as informações necessárias a geração do vídeo, como cor, existência e formação de sprites, estão contidas nesta memória do tipo RAM.

PORTA DE DADOS (ENTRADA E SAÍDA PELA PORTA 98H)

Esta porta é usada para ler ou escrever dados de 8 bits de comprimento (bytes) na VRAM. Para tanto basta informarmos ao VDP através da porta de comandos, se queremos escrever ou ler a VRAM ou ainda um dos diversos registros que compõem o VDP. Se quisermos ler ou escrever na VRAM informamos ao VDP sobre a posição de memória desejada, que ele colocará automaticamente na porta 98H o dado presente na posição de memória selecionada, caso a operação seja de leitura. Se quisermos escrever o processo é o mesmo sendo que agora o VDP espera o dado a ser fornecido pelo Z80 pela porta 98H, e logo a seguir grava-o na posição selecionada. Uma característica interessante é o fato do VDP incrementar automaticamente o registro de endereços após uma operação de leitura ou escrita de dados. Assim sendo, suponha que tenhamos enviado um dado para a posição 0000H da VRAM, após a gravação do dado, o VDP incrementa automaticamente o registro de endereços para a posição 0001H da VRAM, logo um grupo seqüencial de bytes pode ser acessado simplesmente através de uma contínua leitura ou escrita pela porta de dados.

PORTA DE COMANDOS (ENTRADA E SAÍDA PELA PORTA 99H)

Esta porta informa ao VDP a maneira como ele deve interpretar os dados que chegam pela porta 98H. As suas funções são:

1. Ativar o registro de endereços da porta de dados;
2. Ler um registro do VDP, e;
3. Escrever num registro do VDP.

O REGISTRO DE ENDEREÇOS

Este registro indica ao VDP a posição de memória da vídeo RAM (VRAM), a ser lida ou escrita. A posição de memória desejada deverá ser informada ao VDP na forma LSB e MSB. Os bits 7 e 6 do valor MSB é que informam o VDP se desejamos ler ou se desejamos escrever na VRAM. Suponha que quisessemos ler a posição 0000H da VRAM, o procedimento para tanto seria:

```
XOR   A
OUT   (99H),A
OUT   (99H),A
IN    A,(98H)
```

Assim sendo, o valor contido no acumulador após a instrução seria o valor da posição 0000H da VRAM. Agora se quisessemos escrever, teríamos de fazer:

```
XOR   A
OUT   (99H),A
OR    40H
OUT   (99H),A
LD    A,0FFH
OUT   (98H),A
```

Desta forma, com a instrução OR 40H informamos o registro de endereços que desejamos escrever na VRAM. Logo, como se pode notar, se quisermos ler podemos entrar diretamente com o endereço (0000H a 3FFFH), já se quisermos escrever devemos somar 4000H ao endereço pretendido. Um cuidado deve ser tomado pois como uma interrupção do MSX faz uma constante leitura do registro de status do VDP, torna-se necessário desabilitar todas as interrupções antes de escrevermos ou lermos a VRAM.

O REGISTRO DE STATUS (ESTADO) DO VDP

Lendo diretamente a porta 99H obtem-se o estado atual do VDP codificado em bits, como apresenta a figura a seguir.

7	6	5	4	3	2	1	0
Flag F	Flag 5S	Flag C	Número do quinto sprite				

Os bits 0 a 4 indicam o número do sprite (0 a 31) que fez com que o número máximo permitido de 4 sprites por linha fosse excedido. Este indicador se torna necessário porque no MSX só são permitidos no máximo 4 sprites por linha.

O bit 5 indica se houve ou não coincidência de sprites. A coincidência é testada verificando se dois sprites têem um ou mais pixels em comum. No BRASIL como a freqüência é de 60Hz, o teste de coincidência é realizado a cada 16,7ms (1/60), de modo que se houver um movimento muito rápido de sprites pode ser que a sobreposição de dois sprites não seja notada. O valor 1 neste bit indica que houve coicidência de sprites e o valor 0 que não houve.

O bit 6 indica, quando assume o valor 1, que há mais de 4 sprites numa mesma linha, sendo colocado nos bits 0 a 4, o número do sprite a mais.

O bit 7 indica quando no valor 1, que se produziu a última linha ativa que compõem o vídeo. No BRASIL este bit assume o valor 1 a cada 16,7ms. Por oscilar 60 vezes por segundo este sinal é usado pelo Z80 nas suas interrupções, como por exemplo para atualizar o valor do relógio interno.

OS REGISTROS DO VDP

O VDP possui 8 registros, numerados de 0 a 7, somente para escrita. Isto significa que não podemos ler o conteúdo de um registro do VDP. Toda a operação de escrita nos registros envolve somente a porta de comandos. Para escrever num registro devemos enviar primeiramente o dado a ser escrito, logo após devemos informar o número do registro somando a ele o valor 80H e enviando então o resultado para a porta 99H. Todo este procedimento pode ser acompanhado pela figura a seguir, onde escrevemos o valor FFH no registro 3.

```
LD      A,0FFH
OUT     (99H),A
LD      A,03H
OR      80H
OUT     (99H),A
```

Passemos então, a descrição de cada um dos 8 registros:

Registro 0 Neste registro só dois bits desempenham alguma função. O bit 0 indica se o VDP está habilitado a receber dados externos ou não. O valor 0 indica que o VDP está desabilitado e o valor 1 que está habilitado. O bit 1 por sua vez faz parte de um grupo de três que seleciona o modo da tela (gráfico, texto, etc.).

7	6	5	4	3	2	1	0
0	0	0	0	0	0	M3	EV

Registro 1 Este registro é o responsável pelo controle principal do VDP. O bit 0 indica a magnitude dos spri-

tes. Se o seu valor for 0 o tamanho dos sprites será normal, já se for 1, os sprites terão tamanho dobrado. O bit 1 determina se os sprites são do tipo 8x8 bits ou 16x16 bits. Na primeira hipótese o valor do bit 1 será 0 e na segunda 1. O bit 2 não tem função e o seu valor é sempre 0. Os bits 3 e 4 juntamente com o bit 1 do registro 0, indicam ao VDP o modo de tela, da seguinte maneira:

M1	M2	M3	
0	**0**	**0**	**24*32 Modo Texto (Screen 1)**
0	**0**	**1**	**Modo Gráfico (Screen 2)**
0	**1**	**0**	**Modo Multicolor (Screen 3)**
1	**0**	**0**	**24*40 Modo Texto (Screen 0)**

O bit 5 habilita quando no valor 1 o sinal externo de interrupção, e quando no valor 0 desabilita. O bit 6 é usado para habilitar todo o vídeo quando no valor 1, e para desabilitar o sinal de vídeo quando no valor 0. Quando o vídeo é desabilitado a tela permanece com a mesma cor da borda. O bit 7 indica o tipo de VRAM, se o valor for 0 será de 4Kb, já se for 1 será de 16Kb. Aconselho-o a deixar o valor em 1, pois caso contrário o vídeo poderá se tornar ilegível.

7	6	5	4	3	2	1	0
4/16 Kb	**Vid.**	**Int.**	**M1**	**M2**	**0**	**Tam.**	**Mag.**

Registro2 Os quatro bits inferiores assumem valores de 0 a 15 que são depois multiplicados por 400H, indi-

cando então a posição da tabela de nomes nos diversos modos de tela. Os valores possíveis são:

Screen 0 .... 0000B
Screen 1 .... 0110B
Screen 2 .... 0110B
Screen 3 .... 0010B

7	6	5	4	3	2	1	0
0	0	0	0	Tabela de nomes			

Registro 3 Este registro assume um valor que varia de 0 a 255 (FFH), que após ser multiplicado por 40H, fornece o endereço da tabela de cores do um determinado modo de tela. Os valores possíveis são:

Screen 0 .... 00000000B
Screen 1 .... 10000000B
Screen 2 .... 11111111B
Screen 3 .... 00000000B

7	6	5	4	3	2	1	0
Tabela das cores							

Registro 4 O valor contido nos 3 bits inferiores deste registro, quando multiplicado por 800H dá o endereço da tabela de padrões. Os valores possíveis são:

Screen 0 .... 001B
Screen 1 .... 000B

Screen 2 .... 011B
Screen 3 .... 000B

7	6	5	4	3	2	1	0
0	0	0	0	0	Tab. de padrões		

Registro 5 O conteúdo deste registro quando multiplicado por 80H dá o endereço da tabela de atributos dos sprites. Os valores possíveis são:

Screen 0 .... 0000000B
Screen 1 .... 0110110B
Screen 2 .... 0110110B
Screen 3 .... 0110110B

7	6	5	4	3	2	1	0
0	Tabela de atributos dos sprites						

Registro 6 O valor contido neste registro quando multiplicado por 800H nos dá o endereço da tabela dos padrões dos sprites. Os valores possíveis são:

Screen 0 .... 000B
Screen 1 .... 111B
Screen 2 .... 111B
Screen 3 .... 111B

7	6	5	4	3	2	1	0
0	0	0	0	0	Tab. de padr. dos sprites		

Registro 7

Este registro controla a cor dos caracteres e do fundo da tela no modo texto de 24*40 (Screen 0). Os quatro bits inferiores indicam a cor de fundo que pode assumir qualquer valor entre 0 e 15, já os quatro bits superiores controlam a cor dos caracteres, podendo também assumir qualquer valor entre 0 e 15. Nos outros modos de tela os quatro bits inferiores controlam a cor da borda da tela.

7	6	5	4	3	2	1	0
Cor do texto				**Cor do fundo ou da borda**			

OS MODOS DE TELA

O VDP possui quatro modos de tela distintos, cada qual oferecendo um conjunto de características próprias. A partir de agora vamos descrever com pormenores as características de cada um destes modos de tela.

MODO 0 (SCREEN 0)

Este é um modo de texto de 24 linhas e 40 colunas totalizando 960 posições possíveis. A principal característica deste modo é a de só necessitar de duas tabelas: a tabela de nomes e a tabela de padrões. A tabela de nomes se inicia na posição 0000H da VRAM e vai até a posição 03BFH (959D). Nesta tabela estão contidos apenas os códigos ASCII dos caracteres que aparecem na tela. Se por exemplo colocarmos na posição 0000H da VRAM o valor 41H, irá aparecer no lado superior esquerdo o caracter A. Mas se o desenho do um caracter ocupa 8*8 bits, ou seja, 8 bytes, como é que um único byte (no caso do

exemplo 41H) faz aparecer um caracter? A resposta a esta pergunta é simples, existe uma interrupção própria do VDP que lê a tabela de nomes seqüencialmente, e coloca no vídeo os dados presentes na tabela de padrões correspondentes a cada um dos valores lidos da tabela de nomes.

Como já afirmamos este modo oferece a possibilidade de acessarmos 960 (0 a 959) posições de vídeo diferentes. Vamos então ver como escrever um caracter na posição (X,Y) do vídeo. Já que este modo de tela possui 40 colunas se quisermos escrever na posição (0,10), bastanos aplicar a fórmula:

Endereço = 40*Y+X

Neste caso o endereço da VRAM que deveríamos habilitar para escrita é o 400D (0190H). Observe no final deste capítulo um programa para escrever pequenos textos na VRAM que poderá esclarecer melhor os conceitos vistos até aqui.

A tabela de padrões do modo 0 se inicia no endereço 800H (2048D) e apresenta um comprimento de 800H (2048D) bytes. Nesta tabela estão contidos os códigos de formação dos caracteres. Como temos 256 caracteres disponíveis, a tabela é dividida em 256 grupos de 8 bytes cada. Para você ter uma idéia observe como é armazenado o caracter " A " na tabela de padrões dos caracteres.

00100000
01010000
10001000
11111000
10001000
10001000
10001000
00000000

Como você poderá notar os dígitos binários " 1 " é que formam o ca-

racter " A " que aparece na tela, já os bits " 0 " aparecem na tela como pontos transparentes tomando portanto a cor do fundo.

Suponha agora que você queira modificar o grupo de formação de um caracter específico dentro da tabela de padrões dos caracteres. A primeira coisa a fazer é descobrir o endereço do grupo de formação do caracter dentro da tabela de padrões, para tanto aplique a seguinte fórmula:

(CÓDIGO*8) + 2048

Com a fórmula acima você irá obter o endereço decimal do caracter desejado, bastando para isso informar o CÓDIGO ASCII do caracter. Suponha então, que você queira modificar o caracter " A " cujo código ASCII é 65, então:

Endereço = 65*8 + 2048 = 2568D

O resultado acima forneceu o endereço do primeiro dos oito bytes que formam o caracter " A ".

Um cuidado especial deve ser tomado, quando você alterar a tabela de padrões dos caracteres, pois um simples comando SCREEN 0 irá apagar todo o seu trabalho. Para evitar este contratempo aconselho-o a formar a sua tabela redefinida na RAM juntando a ela uma pequena sub-rotina que transfira a nova tabela para a VRAM. A principal vantagem de colocar a nova tabela na RAM é a de permitir que ela seja gravada com o comando BSAVE normal do MSX BASIC. Para transferir dados da RAM para a VRAM você deve usar várias instruções OTIR como poderá verificar no primeiro exemplo no final deste capítulo quando transferimos uma cadeia de caracteres da RAM para a VRAM.

MODO 1 (SCREEN 1)

Este modo é também para texto, apresentando porém, uma outra característica importante: a possibilidade de usarmos sprites. A tabela de nomes se inicia no endereço 1800H e apresenta um comprimento total de 300H (768) bytes, devido ao fato de possuir 24 linhas com 32 colunas (32*24=768). A fórmula para calcular o endereço de impressão de um caracter na tela nas coordenadas (X,Y) é então:

Endereço = 6144 + 32*Y + X

Se quisermos imprimir o caracter " A " na posição (0,10) devemos mandar o código 41H para o endereço 6464D da VRAM. Note que a coordenada X está compreendida entre os valores 0 e 31 e a coordenada Y entre os valores 0 e 23.

A tabela de padrões dos caracteres ocupa 2048 bytes como no modo 0, iniciando no endereço 0000H e terminando no endereço 07FFH, sendo dividida em 256 blocos de 8 bytes de comprimentos também. O endereço do grupo de formação de um determinado caracter é obtido agora, do seguinte modo:

Endereço = CÓDIGO*8

Assim sendo, o grupo de formação do caracter " A " se encontra a partir do endereço:

Endereço = 65*8 = 520

Da mesma forma que no modo 0, aconselho-o a preservar as alterações, que porventura você venha a fazer, na tabela de padrões dos caracteres na memória RAM.

Além das tabelas de nome e padrões dos caracteres existem as tabe-

las de cor e de sprites, que veremos mais adiante quando estudarmos os sprites.

A tabela de cor inicia-se na posição 2000H (8192) da memória VRAM e apresenta um comprimento total de 32 bytes. A função desta tabela é indicar a cor dos pixeis (pontos na tela) 1 e 0. Cada byte da tabela de cor indica assim a cor de 8 códigos ASCII sendo a cor dos pontos acesos (pixel 1) determinada pelos quatro bits superiores do byte da tabela de cor, e a cor dos pixeis apagados (pixel 0) determinada pelos quatro bits inferiores do mesmo byte da tabela de cores. Deste modo o primeiro byte da tabela de cores determinará as cores dos caracteres ASCII 0 a 7, o segundo byte determinará as cores dos caracteres 8 a 15 e assim por diante. Para você entender bem como funciona este processo, coloque o VDP no modo de tela 1 com o comando SCREEN 1, e depois digite o seguinte programa em BASIC:

```
10 FOR A%=&H2000 TO &H201F
20 VPOKE A%,255*RND(- TIME)
30 NEXT A%
```

Rode o programa com o comando RUN e observe a "bagunça" de cores da sua tela. Para voltar ao normal aperte a tecla F6.

MODO 2 (SCREEN 2)

O modo 2 é o que oferece mais recursos e mais dores de cabeça ao programador, por ser mais difícil de se programar que os dois anteriores. A tabela de nomes ocupa 768 bytes de VRAM, iniciando no endereço 1800H e terminando no 1AFFH. O modo de disposição desta tabela é igual ao do modo 1 visto anteriormente. Entretanto há uma grande diferença na maneira desta tabela definir a imagem que vai aparecer na tela no modo 2 em relação aos modos 0 e 1. Acontece que nos modos 0 e 1 bastava-nos informar a tabela de nomes, o códi-

go do caracter que desejávamos imprimir na tela, que logo aparecia aceso na posição especificada. No modo 2 a combinação do valor da posição dentro da tabela de nomes juntamente com a posição dentro da tabela de padrões, é que define a imagem que vai aparecer na tela. Embora o conceito acima lhe pareça complicado demais vamos prosseguir nas apresentações das características do modo 2, e logo voltaremos a explicação de como os caracteres são impressos na tela de alta resolução.

A tabela de padrões é a maior de todos os modos de tela ocupando 6144 bytes (1800H) ocupando desde o endereço 0000H ao 17FFH da VRAM. Embora a sua estrutura seja a mesma dos modos 0 e 1, ela não é inicializada com grupos de formação dos caracteres ASCII, mas sim preenchida com zeros. Ao inicializar o modo gráfico, o MSX repete três vezes a seqüência de códigos dos caracteres ASCII na tabela de nomes, assim sendo a tabela de nomes fica dividida em 3 blocos de 256 bytes. Já a tabela de padrões é inicializada com o valor zero em toda a sua extensão de 6Kb, onde os 2Kb iniciais estão endereçados pelos códigos dos caracteres da primeira parte das três partes de 256 bytes que compõem a tabela de nomes. Da mesma forma os segundos 2Kb da tabela de padrões estão endereçados pelo segundo grupo de 256 bytes da tabela de nomes, e os terceiros 2Kb da tabela de padrões estão endereçados pelo terceiro grupo de 256 bytes da tabela de nomes. A tabela de cores também ocupa 6144 bytes iniciando-se na posição 2000H e terminando na posição 37FFH. Para cada byte da tabela de padrões existe um byte equivalente na tabela de cores, onde os quatro bits menos significativos indicam a cor dos pixeis " 0 " e os quatro bits mais significativos indicam a cor dos pixeis " 1 ". Para que você possa entender o mecanismo de impressão na tela de alta resolução, leia atentamente o próximo paragráfo.

Os primeiros 2048 bytes da tabela padrões definem as imagens correspondentes aos primeiros 256 bytes da tabela de nomes, e assim por diante para os segundos e terceiros 2Kb da tabela de padrões que se correspondem aos segundos e terceiros blocos de 256 bytes da ta-

bela de nomes. Para tirar todas as dúvidas acompanhe o programa BASIC a seguir:

```
 10 SCREEN 2
 20 REM DEFINE O CARACTER 0 COMO A
 30 FOR A%=0 TO 7:READ B$:VPOKE A%,VAL ("&B"+B$):NEXT
    A%
 40 REM IMPRIME A PRIMEIRA LINHA DA TELA COM A LETRA
    A
 50 FOR A%=&H1800 TO &H181F:VPOKE A%,0:NEXT A%
 60 REM ESTABELECE A COR DOS CARACTERES IMPRESSOS
 70 FOR A%=&H2000 TO &H37FF:VPOKE A%,&HF1:NEXT A%
 80 REM LOOP DE ESPERA
 90 IF INKEY$=" " THEN 90 ELSE SCREEN 0
100 DATA 00111000
110 DATA 11001100
120 DATA 11001100
130 DATA 11111100
140 DATA 11001100
150 DATA 11001100
160 DATA 11001100
170 DATA 00000000
```

Rode o programa acima com o comando RUN e observe o resultado.

MODO 3 (SCREEN 3)

O modo 3 oferece gráficos de baixa resolução com 16 cores e possibilidade de uso de sprites. Uma característica interessante deste modo é a ausência de uma tabela de cores. A informação da cor é dada pela tabela de padrões, como veremos.

A tabela de nomes inicia na posição 0800H e termina na posição

0AFFH, tendo portanto uma extensão de 768 bytes. Ao inicializar este modo o MSX divide a tabela de nomes em 24 blocos de 32 bytes, representando as 24 linhas e 32 colunas da tela. Os blocos são iguais 4 a 4, isto quer dizer que os blocos 1, 2, 3 e 4 são iguais mas distintos dos blocos 5, 6, 7 e 8 que são iguais entre si, e assim por diante até os blocos 21, 22, 23 e 24. A figura seguinte ilustra este processo bem como os valores que são atribuídos a cada bloco.

Linha da Tela	Código na coluna 0	Código na coluna 31
0 a 3	00H	1FH
4 a 7	20H	3FH
8 a 11	40H	5FH
12 a 15	60H	7FH
16 a 19	80H	9FH
20 a 23	A0H	BFH

Agora que vimos como é inicializada a tabela de nomes, vamos entender como se acessa a entrada da correspondente tabela de padrões, para que efetivamente apareça uma imagem no vídeo.

A tabela de padrões inicia no endereço 0000H e termina no endereço 07FFH da VRAM, ocupando portanto 2048 bytes. Esta tabela contém a informação sobre a imagem que vai aparecer na tela como também a informação das cores.

Byte 0	AAAABBBB
Byte 1	CCCCDDDD

A	B
C	D

No modo 3 cada pixel é formado por uma matriz de 4*4 bits, isto quer dizer que cada pixel deste modo corresponde a um grupo de 16 pixeis do modo gráfico por exemplo. A figura anterior ilustra a correspondência entre a tabela de padrões e a tela no modo 3.

Por causa da baixa resolução é necessário somente um byte da tabela de padrões para definir dois pixeis 4*4 na tela. Deste modo para definirmos as cores de um quadrado de 4 pixeis, necessitamos de dois bytes como mostra a figura anterior. Assim sendo, cada dois bytes dos oito primeiros que compõem a tabela de padrões, define uma imagem que pode aparecer na tela como conseqüência de uma entrada na tabela de nomes, que vai definir o local da imagem na tela.

Byte	
0	**Linhas 0,4,8,12,16,20**
1	
2	**Linhas 1,5,9,13,17,21**
3	
4	**Linhas 2,6,10,14,18,22**
5	
6	**Linhas 3,7,11,15,19,23**
7	

Se você não conseguiu entender o funcionamento, suponha que você queira imprimir na primeira região da tela, que compreende as linhas 0, 1, 2 e 3. As cores da primeira coluna (coluna 0) de 8 bits de comprimento são definidas pelos oito primeiros bytes da tabela de padrões, as cores da segunda coluna são definidas pelo segundo grupo de 8 bytes da tabela de padrões, e assim por diante. Para tirar todas as dúvidas digite e rode o seguinte programa BASIC:

```
 10 SCREEN 3
 20 REM DEFINE AS COLUNAS A SEREM PREENCHIDAS
 30 FOR A%= 0 TO &H1F
 40 REM DÁ COR AS 4 LINHAS DE CADA COLUNA
 50 FOR B%= 0 TO 6 STEP 2
 60 REM COLOCA AS CORES
 70 VPOKE A%*8+B%,&HF4
 80 VPOKE A%*8+B%+1,&H4F
 90 NEXT B%
100 NEXT A%
110 REM LOOP DE ESPERA PARA PODER APRECIAR A TELA
120 IF INKEY$=" " THEN 120 ELSE SCREEN 0
```

Rode o programa e observe o resultado. Se você quiser alterar as cores mude os valores F4H e 4FH nas linhas 70 e 80.

OS SPRITES

Os sprites podem ser definidos como caracteres definidos pelo usuário e que apresentam a capacidade de se movimentarem pela tela, bastando para isso definir as suas coordenadas dentro da tela. Os sprites apresentam também a capacidade de se sobreporem a caracteres já existentes sem contudo alterá-los. O VDP pode controlar até 32 sprites em todos os modos exceto no modo texto de 24 linhas por 40 colunas (modo 0). O tratamento dos sprites é igual em todos os modos, de maneira que todas as características a serem vistas são válidas em todos os modos que permitam sprites.

A tabela de atributos dos sprites ocupa 128 bytes iniciando no endereço 1B00H e terminando no endereço 1B7FH. Esta tabela contém 32 grupos de 4 bytes, onde cada grupo é o responsável pela definição das características de um determinado sprite. O primeiro bloco de 4 bytes contém as informações pertinentes ao sprite "0", e assim por diante

até o sprite " 31 ". As informações contidas em cada um destes blocos se apresentam como na figura a seguir:

	7	6	5	4	3	2	1	0
Byte 0	**Posição Vertical do sprite na tela**							
Byte 1	**Posição Horizontal do sprite na tela**							
Byte 2	**Modelo do sprite**							
Byte 3	**EC**	**0**	**0**	**0**	**Cor da sprite**			

Nesta tabela o byte 0, indica a coordenada Y do pixel situado no lado esquerdo do topo do sprite. Os valores possíveis vão de -1 (FFH) a 190 (BEH), onde o primeiro coloca o sprite no topo da tela e o segundo na última linha da tela. Os valores menores que -1 podem ser usados para simular um efeito de corte do sprite. Um cuidado todo especial deve ser tomado para que não se coloque neste byte o valor 208 (D0H), pois o VDP passa a ignorar todos os grupos posteriores de atribuição de sprites.

O byte 1 especifica, por sua vez, a coordenada X do pixel situado no lado esquerdo superior do sprite. Os valores possíveis já são mais coerentes com os usados no BASIC indo desde o valor 0 ao 255 (00H a FFH).

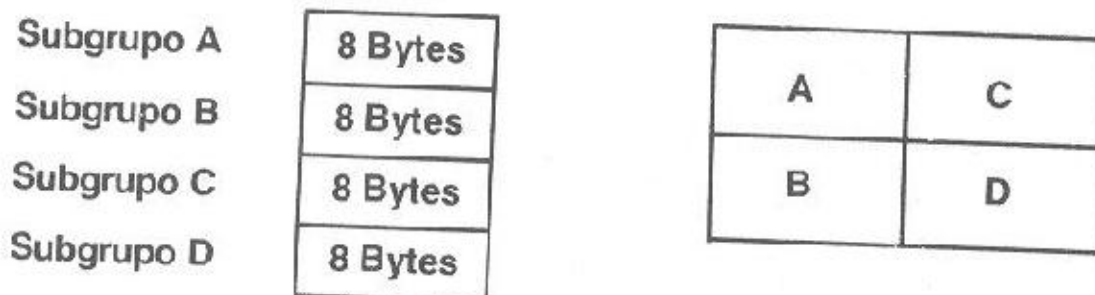
O byte 2 seleciona um dos trinta e dois modelos possíveis para o modelo de sprite, que se encontram na tabela de modelos dos sprites. Se o tamanho dos sprites for definido como sendo de 16*16 bits, cada modelo de sprite irá ocupar 32 bytes na tabela de modelos, já se o tamanho for de 8*8 bits cada modelo irá ocupar 8 bytes.

Os quatro bits menos significativos do byte 3 determinam a cor dos pixeis " 1 " do modelo de sprite, já os pixeis " 0 " são sempre atribuídos com a cor transparente. O bit 7 chamado de "early clock" apresenta

normalmente o valor 0, porém quando colocamos o seu valor em 1, o VDP provoca um deslocamento de 32 pixeis para a esquerda do sprite do plano em questão. Ao colocarmos este bit em 1 deslocamos o sprite para a posição X-32,Y em vez de deixá-lo na posição X,Y.

A tabela de modelos de sprites ocupa 2048 bytes iniciando no endereço 3800H e terminando no endereço 3FFFH. Esta tabela contém 256 grupos de 8 bytes, que definem assim, 256 modelos de sprites de tamanho 8*8 bits. Se optarmos pelos sprites 16*16 bits o número de modelos possíveis baixa para 64, sendo o modelo do sprite definido por um grupo de 32 bytes. A figura a seguir ilustra este último caso:

Subgrupo A	**8 Bytes**
Subgrupo B	**8 Bytes**
Subgrupo C	**8 Bytes**
Subgrupo D	**8 Bytes**

A	**C**
B	**D**

Após a apresentação das características do VDP vamos apresentar alguns exemplos para fixar melhor os conceitos vistos até agora.

1º. Exemplo: Como imprimir um texto no modo 0

```
1000    ORG     0E000H
1010    DI
1020    LD      HL,0000H
1030    LD      BC,03C0H
1040    CALL    CLS
1050    LD      HL,(0F7F8H)
1060    LD      B,(HL)
1070    INC     HL
1080    LD      E,(HL)
1090    INC     HL
```

```
1100          LD      D,(HL)
1110          PUSH    BC
1120          PUSH    DE
1130          LD      A,10
1140          LD      L,A
1150          LD      H,00H
1160          LD      E,A
1170          LD      D,00H
1180          LD      B,39
1190  LOOP:   ADD     HL,DE
1200          DJNZ    LOOP
1210          LD      A,10
1220          LD      E,A
1230          LD      D,00H
1240          ADD     HL,DE
1250          POP     DE
1260          POP     BC
1270          CALL    VRAM
1280          EX      DE,HL
1290          LD      C,98H
1300          OTIR
1310          EI
1320          RET
1330  CLS:    LD      A,L
1340          OUT     (99H),A
1350          LD      A,H
1360          OR      40H
1370          OUT     (99H),A
1380  LOOP1:  LD      A,20H
1390          OUT     (98H),A
1400          DEC     BC
1410          LD      A,B
1420          OR      C
1430          RET     Z
1440          JR      LOOP1
```

```
1450   VRAM:   LD      A,L
1460           OUT     (99H),A
1470           LD      A,H
1480           OR      40H
1490           OUT     (99H),A
1500           RET
```

O funcionamento do programa acima é bem simples tendo em vista que o texto a ser impresso é fornecido pelo argumento da função USR do BASIC. Da linha 1020 a 1040 temos a entrada de parâmetros para uma sub-rotina (CLS) que limpa toda a tela preenchendo-a com espaços. Da linha 1050 a linha 1100, pegamos os parâmetros da função USR colocando no registro B o comprimento da string a ser impressa, e no par DE o endereço de memória onde se encontra o primeiro caracter da string. Na linha 1130 estabelecemos a coordenada Y e calculamos o endereço correspondente a esta coordenada, multiplicando-a por 40 (da linha 1140 a 1200). Na linha 1210 entramos com a coordenada X e somamos o seu valor ao endereço achado anteriormente. Após a etapa de cálculo para achar o endereço da VRAM correspondente as coordenadas (X,Y), passamos então a impressão do texto na tela, usando para isso a instrução OTIR. Finalmente nas linhas 1310 e 1320 preparamos a volta ao BASIC. Para testar este programa compile-o e rode-o através do seguinte programa em BASIC:

```
10 SCREEN 0
20 DEFUSR=&HE000
30 A$=USR("COMPUTADOR MSX")
40 IF INKEY$=" " THEN 40 ELSE END
```

Estude o programa e faça as alterações que achar conveniente. Desde já eu posso lhe afirmar que se você mudar de lugar a parte que pega os parâmetros da função USR, poderá economizar dois PUSH's e dois POP's.

2º. Exemplo: Como traçar uma linha no modo 2

```
1000            ORG     0E000H
1010            DI
1020            LD      HL,2000H
1030            LD      BC,1800H
1040            CALL    COR
1050            LD      HL,(0F7F8H)
1060            LD      DE,0008H
1070            LD      B,32
1080  LOOP1:    LD      A,0FFH
1090            CALL    VRAM
1100            ADD     HL,DE
1110            DJNZ    LOOP1
1120            EI
1130            RET
1140  COR:      LD      A,L
1150            OUT     (99H),A
1160            LD      A,H
1170            OR      40H
1180            OUT     (99H),A
1190  LOOP2:    LD      A,0F1H
1200            OUT     (98H),A
1210            DEC     BC
1220            LD      A,B
1230            OR      C
1240            RET     Z
1250            JR      LOOP2
1260  VRAM      PUSH    AF
1270            LD      A,L
1280            OUT     (99H),A
1290            LD      A,H
1300            OR      40H
1310            OUT     (99H),A
1320            POP     AF
```

```
1330        OUT        (98H),A
1340        RET
```

Este programa é bem simples como você poderá verificar. Nas linhas 1020 a 1040 preenchemos a tabela de cores com o valor F1H, para podermos desenhar linhas em branco (FH) em fundo preto (1H), para isso chamamos a sub-rotina cor. Da linha 1050 a 1070 pegamos os parâmetros da função USR onde o byte menos significativo indica a posição do pixel dentro da linha e, o byte mais significativo a linha do pixel (0 a 23). O registro B recebe o número máximo de colunas e o registro DE recebe o incremento para imprimimos os pixeis numa mesma linha. A impressão é feita pela sub-rotina VRAM como você pode observar. Para rodar este programa, compila-o e digite o programa BASIC a seguir:

```
10  SCREEN 2
20  DEFUSR=&HE000
30  A=USR(&HB00)
40  IF INKEY$="  " THEN 40 ELSE END
```

Rode o programa acima e observe a impressão de uma linha na tela de alta resolução. Se você quiser enfeitar mais, substitua a linha 30 pela linha abaixo:

```
30  FOR A%=&HB00 TO &HB07:A=USR(A%):NEXT A%
```

3º. Exemplo: Como criar e movimentar sprites

```
1000        ORG        0E000H
1010        DI
1020        LD         HL,SPRITE
1030        LD         B,08H
1040        LD         C,98H
1050        LD         DE,3800H
1060        CALL       VRAM
```

```
1070            LD      HL,ATRIB
1080            LD      B,04H
1090            LD      C,98H
1100            LD      DE,1B00H
1110            CALL    VRAM
1120            LD      HL,ATRIB
1130            LD      DE,0F600H
1140            LD      BC,0004H
1150            LDIR
1160  LOOP:     LD      BC,0A00H
1170  ESPERA:   DEC     BC
1180            LD      A,B
1190            OR      C
1200            JR      NZ,ESPERA
1210            CALL    TECLA
1220            BIT     0,A
1230            JR      Z,FIM
1240            BIT     7,A
1250            JR      Z,AUMX
1260            BIT     4,A
1270            JR      Z,DIMX
1280            BIT     5,A
1290            JR      Z,DIMY
1300            BIT     6,A
1310            JR      Z,AUMY
1320            JR      LOOP
1330  AUMX:     LD      A,(0F601H)
1340            INC     A
1350            LD      (0F601H),A
1360            LD      HL,0F601H
1370            LD      B,01H
1380            LD      C,98H
1390            LD      DE,1B01H
1400            CALL    VRAM
1410            JR      LOOP
```

```
1420  DIMX:   LD     A,(0F601H)
1430          DEC    A
1440          LD     (0F601H),A
1450          LD     HL,0F601H
1460          LD     B,01H
1470          LD     C,98H
1480          LD     DE,1B01H
1490          CALL   VRAM
1500          JR     LOOP
1510  AUMY:   LD     A,(0F600H)
1520          INC    A
1530          LD     (0F600H),A
1540          LD     HL,0F600H
1550          LD     B,01H
1560          LD     C,98H
1570          LD     DE,1B00H
1580          CALL   VRAM
1590          JR     LOOP
1600  DIMY:   LD     A,(0F600H)
1610          DEC    A
1620          LD     (0F600H),A
1630          LD     HL,0F600H
1640          LD     B,01H
1650          LD     C,98H
1660          LD     DE,1B00H
1670          CALL   VRAM
1680          JR     LOOP
1690          JR     LOOP
1700  FIM:    EI
1710          RET
1720  TECLA:  LD     A,0F8H
1730          OUT    (0AAH),A
1740          IN     A,(0A9H)
1750          RET
1760  VRAM:   EX     DE,HL
```

```
1770            LD      A,L
1780            OUT     (99H),A
1790            LD      A,H
1800            OR      40H
1810            OUT     (99H),A
1820            EX      DE,HL
1830            OTIR
1840            RET
1850  SPRITE:   DEFB    00111000B
1860            DEFB    00111000B
1870            DEFB    00010000B
1880            DEFB    00111000B
1890            DEFB    01010100B
1900            DEFB    00010000B
1910            DEFB    00101000B
1920            DEFB    01000100B
1930  ATRIB:    DEFB    100,100,0,0FH
```

O programa acima é o mais complexo feito até agora, vamos então examiná-lo por partes. Da linha 1020 a 1150 definimos o modelo do sprite (linha 1020 a 1060), definimos os atributos iniciais (linhas 1070 a 1110) e, transferimos a tabela de atributos para uma região da RAM para assim criarmos um bloco que descreva permanentemente as coordenadas do sprite na tela. Da linha 1160 a 1320 temos o chamado loop principal do programa, que é o encarregado de ler o teclado e de tomar as devidas decisões mediante os resultados dessa leitura. Assim se a tecla lida foi a do cursor para cima o programa desvia-se para a rotina AUMY, que atualiza então a nova coordenada (Y), e do mesmo modo para as rotinas DIMY, AUMY e DIMX. Para voltar ao BASIC fazemos um teste na linha 1220, onde verificamos se a tecla de espaço foi pressionada. Você pode estranhar o fato de colocamos uma sub-rotina de transferência de atributos para a VRAM, em cada rotina de atualização de coordenadas, quando poderíamos colocar só uma no loop principal do programa. A razão para essa aparente discrepância se deve ao fato de só podermos alterar uma coordenada de cada vez

sem perda de sincronismo, se não fizessemos isto o sprite apareceria na tela piscando. Se você desejar que o sprite se mova rápido modifique o valor de BC na linha 1160 para um valor menor.

Para testar o programa acima compile-o e digite o programa BASIC a seguir:

```
10  SCREEN 2,0
20  DEFUSR=&HE000
30  A=USR(0)
40  END
```

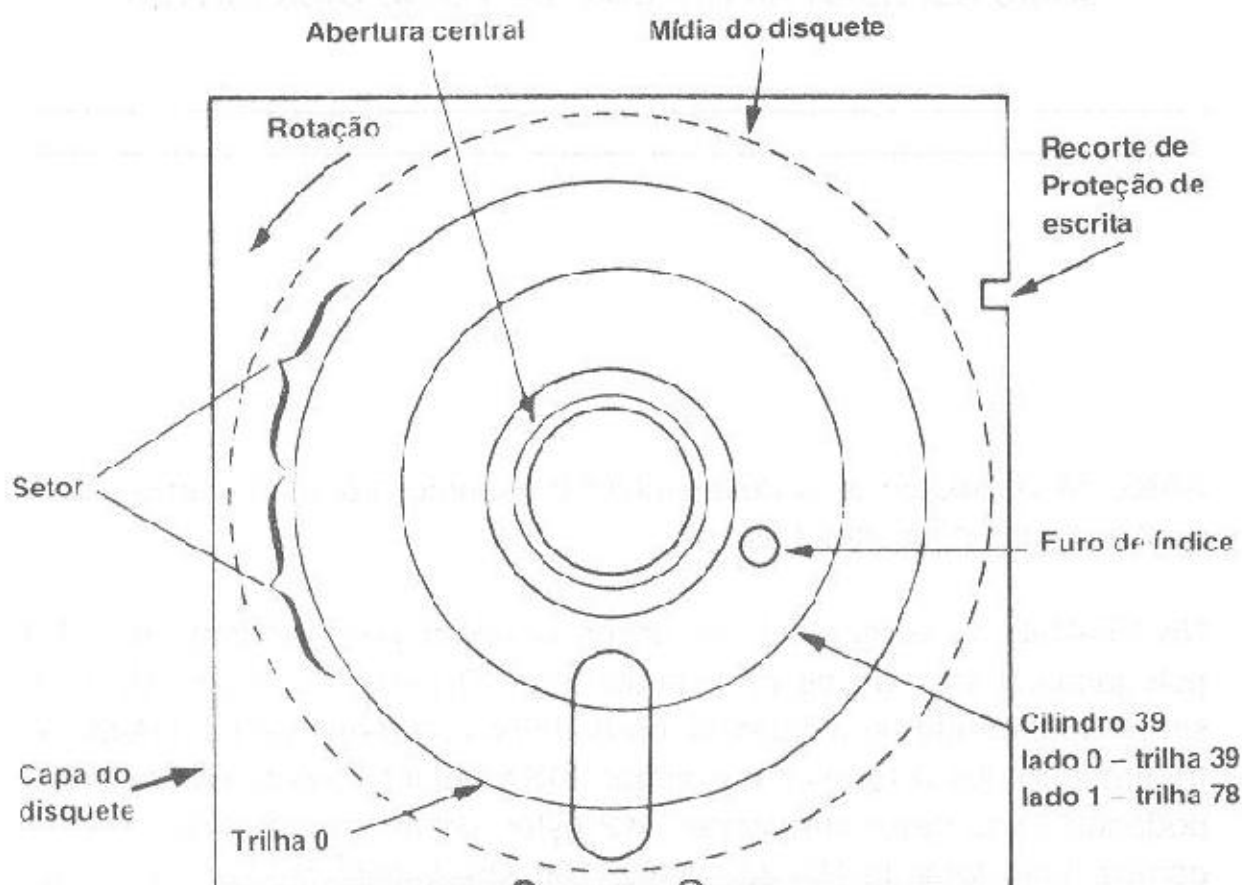
Com este capítulo encerramos o estudo sobre o MSX na sua versão mais simples, ou seja, sem disk drive. O próximo capítulo será dedicado a todos aqueles que possuem drive e se encontram desolados por não encontrarem as informações necessárias.

CAPÍTULO 18

COMO ACESSAR AS ROTINAS DE I/O DO DISK DRIVER

Antes de começarmos a desenvolver o assunto proposto, vamos apresentar alguns conceitos básicos.

No BRASIL os fabricantes de drives optaram pelo modelo de 5 1/4 polegadas. Estes drives podem acessar 40 trilhas se forem de face simples (somente uma cabeça), ou 80 trilhas se forem de face dupla. A formatação dos disquetes no padrão MSX habilita 9 setores por trilha, podendo cada setor armazenar 512 bytes de informação. Assim chegamos a um total de 184.320 bytes (180 Kb) disponíveis para gravação de dados no modelo de face simples e 368.640 bytes (360 Kb) no modelo de face dupla. Na verdade o espaço líquido para escrita é menor em virtude da trilha 0 (setores 0 a 8) ser utilizada para armazenar a rotina de boot (partida) do drive, um mapa dos setores que constituem os diversos arquivos e, finalmente o diretório que ocupa os setores 5 a 8 da trilha 0 se o drive for simples, e os setores 5 a 11 se o drive for de dupla face. Como se pode notar o espaço útil para escrita cai para 179.712 bytes no modelo simples, e para 362.496 bytes no modelo de

face dupla. Tanto num modelo como noutro, a rotina de partida ocupa o setor 0 da trilha 0. Esta rotina é que informa ao sistema sobre as características do disco em questão: se ele é simples ou se é dupla face. Como no drive de dupla face os arquivos são gravados usando-se as duas faces simultaneamente, você não conseguirá ler os arquivos gravados por este modelo, caso o seu drive seja de face simples. A figura a seguir ilustra um modelo de disquete de 5 1/4 polegadas:

COMO LER O DIRETÓRIO

Já sabemos que no modelo simples o diretório encontra-se entre os setores 5 a 8, e no modelo de dupla face entre os setores 5 a 11. O diretório contém por assim dizer as informações pertinentes ao arquivo, como nome, o setor inicial a partir do qual começou a ser gravado o

arquivo, como também o tamanho do arquivo em bytes, além de outras informações como a data por exemplo, que no momento não nos interessam. Vamos então mostrar um exemplo prático em BASIC de como ler as informações contidas no diretório:

```
100  CLS:KEYOFF
110  GOSUB 1000
120  Z=5
130  CLS:A$=DSKI$(0,Z)
140  FOR A=EN TO (EN+511) STEP 32
150  IF PEEK(A)=0 THEN Z=ZM: GOTO 310
160  FOR B=0 TO 7
170  LOCATE(2+B):PRINT CHR$(PEEK(A+B) );
180  NEXT B
190  PRINT ".";
200  FOR B=8 TO 10
210  PRINT CHR$(PEEK(A+B) );
220  NEXT B
230  IF ZM=11 THEN SI=2*(PEEK(A+26)+256*PEEK(A+27) )+8
     ELSE SI=PEEK(A+26)+256*PEEK(A+27)+7
240  TM=PEEK(A+28)+256*PEEK(A+29)
250  SF=SI+INT(TM/512)-1:FG=0
260  IF TM<=&H7FFF THEN IF (TM MOD 512) <> 0
     THEN SF=SF+1:FG=1
270  IF TM>&H7FFF THEN FM=65536-TM:IF (FM MOD 512)
     <>0 THEN SF=SF+1:FG=1
280  LOCATE 18:PRINT SI;:LOCATE 25:
     PRINT SF;:LOCATE 31:PRINT TM
290  NEXT A
300  IF INKEY$=" " THEN 300
310  Z=Z+1:IF Z=ZM+1 THEN END ELSE GOTO 130
1000 REM SUB-ROTINA PARA DETECTAR SE O DISCO FOI
     GRAVADO POR UM DRIVE DE DUPLA OU DE SIMPLES
1010 EN=PEEK(&HF351)+256*PEEK(&HF352)
1020 A$=DSKI$(0,0)
```

```
1030 IF (PEEK(EN+21) AND 1)=0 THEN ZM=8 ELSE ZM=11
1040 RETURN
```

O programa acima é bem simples, e com ele o leitor poderá entender o mecanismo de armazenamento das informações no diretório. Para que você possa entender onde estão localizadas as informações necessárias, obtidas acima através do comando PEEK, observe o "dump" criado sobre o arquivo SOLXDOS.SIS presente no diretório do disco de sistema.

```
53  4F  4C  58  44  4F  53  20        SOLXDOS
53  49  53  00  00  00  00  00        SIS......
00  00  00  00  00  00  00  0Q        ..........
06  0F  02  00  00  21  00  00        .......!..
```

O dump acima foi criado por um **zapper** nome que se dá aos programas que "vasculham" o disco) de minha autoria, do qual me orgulho muito pois além de permitir a alteração de qualquer setor com editor full-screen, apresenta outras facilidades como testes de hardware, entre eles a velocidade do drive. Veja mais informações sobre este programa no final do livro. Bom deixemos de comerciais e voltemos ao assunto que mais interessa. Como você pode notar os 8 primeiros bytes (0 a 7) informam o nome do arquivo, do byte 9 ao 11 temos a extensão do nome do arquivo (SIS, BAS, BIN, etc.), nos bytes 26 e 27 temos os valores menos significativos e mais significativos respectivamente, do valor do setor inicial a partir do qual está gravado o arquivo. O valor contido neste byte não informa diretamente o número do setor inicial, se você quiser obter o número correto terá de somar 7 ao valor obtido se o seu drive for simples, ou multiplicar o valor obtido por 2 e somar 8 se o seu drive for de face dupla. Observe que o programa em BASIC acima faz esse teste. Os bytes 28 e 29 informam o tamanho do arquivo na forma LSB e MSB. Os bytes 30 e 31 não são usados. Como você pode notar cada arquivo gasta 32 bytes dos setores do diretório para armazenar todas as suas características.

AS PORTAS DE ACESSO AO DRIVE

Pelo manual da MICROSOFT as portas de D0H a D7H foram reservadas para a interface controladora do disk drive. Se seu drive for da MICROSOL ou EXPAND todas as informações a serem expostas irão se aplicar, infelizmente o mesmo não ocorre se o seu drive for da SHARP, que pelo pouco que eu examinei faz os seus acessos por endereços de memória e não através do portas.

A única porta de interesse agora é a porta D4H, que seleciona o drive (A ou B) e liga ou desliga o motor do drive selecionado. Outra porta útil, mas que não veremos aqui com maiores detalhes, é a porta D0H, que informa por exemplo se o buraco de index do disquete está sobre a luz detetora de índice ou não.

No parágrafo anterior afirmei que a porta D4H era de grande interesse sobretudo por permitir o controle sobre o motor do drive. Vejamos como isso ocorre. Tente digitar o seguinte comando direto:

OUT &HD4,&B00100001

Se pode observar então, que o motor do drive A se acionou e a luz indicadora se acendeu. Digite agora a seguinte sentença:

OUT &HD4,&B00000001

Agora o motor parou e a luz indicadora permaneceu acesa. Assim você deve ter percebido que o bit 5 do dado a ser enviado para a porta D4H controla o motor do drive selecionado. Digite agora:

OUT &HD4,&B00000000

Observe que a luz indicadora do drive se apagou. O bit 0 é o responsável pela seleção do drive A. Se você tem dois drives digite o coman-

do a seguir.

OUT &HD4,&B00100010

Como você já devia esperar este comando selecionou o drive B e ligou o seu motor.

Usando estes conceitos você já pode por exemplo parar o drive, terminando assim com o velho problema de quando se carrega programas em linguagem de máquina (jogos) o drive permanecer ligado. Para desligar o drive proceda da seguinte maneira:

1º. Descubra os endereços inicial, final e de execução.

2º. Após o endereço final dê os seguintes POKES:

POKE XXXX+1,&HAF
POKE XXXX+2,&HD3
POKE XXXX+3,&HD4
POKE XXXX+4,&HC3
POKE XXXX+5,&Hyy
POKE XXXX+6,&HYY

onde: XXXX = endereço final do programa
YYyy = endereço de execução, onde yy é o LSB e YY o MSB.

3º. Grave o programa com o mesmo endereço inicial, com o endereço final dado por XXXX+6, e com o endereço de execução dado por XXXX+1.

COMO ACESSAR AS ROTINAS DE I/O EM DISK BASIC

Este tópico se destina a explicar como se deve acessar as rotinas de I/O sem recorrer ao MSXDOS. Em princípio só temos uma rotina na ROM para acessar o drive, é ela:

Nome: PHYDIO

Entrada: B=número de setores a serem lidos ou escritos
C=parâmetro de formatação (F8H)

DE=Número do setor a ser carregado ou escrito
HL=Endereço inicial da RAM que vai receber ou gravar os dados
A=drive selecionado (0=A)

Se o flag do carry for igual a 1, a operação será de escrita, se 0 será de leitura.

Endereço: 0144H

A rotina acima não tem entretanto grande interesse a não ser que desejásssemos proteger programas em disco. Felizmente existe uma sub-rotina na RAM que funciona de modo semelhante a famosa rotina da BDOS do MSXDOS. Vamos então às suas características. Ao acessar esta sub-rotina devemos informá-la que tipo de operação desejamos, colocando o código da operação desejada no registro C. Alguns parâmetros também são passados pelo par DE, como veremos. A Seguir temos um quadro resumo das principais operações desta rotina:

N.	OPERAÇÃO	ENTRADA	SAÍDA	OBSERVAÇÕES
0FH	Abrir Arquivo	DE=FCB	A=achou ou não	Muito usado para verificar se já existe o arquivo. Além de abrir o arquivo.
10H	Fecha Arquivo	DE=FCB	A=achou ou não	Esta operação deve ser sempre executada ao final de utilização de qualquer arquivo.
13H	Elimina um Arquivo	DE=FCB	A=achou ou não	Esta operação deve ser executada com todo o cuidado, pois uma vez apagado não há como recuperar um arquivo.
14H	Leitura Seqüencial	DE=FCB	A=Código de erro	Esta operação carrega 128 bytes do arquivo para o DMA.
15H	Escrita Seqüencial	DE=FCB	A=Código de erro	Esta operação grava os 128 bytes contidos no DMA.
16H	Criar Arquivo	DE=FCB	A=Código do diretório	Esta operação cria um arquivo sem verificar se ele já existe.
1AH	Indica endereço ao DMA	DE=DMA	Nenhuma	Esta operação deve ser usada para especificar o DMA.

Na figura acima vimos dois termos desconhecidos: o FCB e o DMA. O

termo FCB, vem do inglês "File control block" e nada mais é que um endereço de memória onde o usuário guarda as informações sobre os arquivos com os quais deseja trabalhar. Esta área ocupa 37 bytes, onde somente os doze primeiros bytes são manipulados pelo usuário. O byte 0 indica o drive selecionado (1=A, 2=B), já do byte 1 ao 8 está armazenado o nome do arquivo, e do byte 9 ao 11 a sua extensão. O termo DMA, vem do inglês "Direct memory acess" e nada mais é que o endereço de memória a partir do qual são guardados os dados a serem lidos ou escritos. Vale notar que o comprimento do DMA é de 128 bytes, e que apesar de cada setor do disco comportar 512 bytes, o sistema só grava 128 bytes por vez, repetindo portanto a operação mais três vezes até completar o setor.

Para acessar todas estas operações você deve colocar no registro C a operação desejada e depois chamar a sub-rotina da BDOS com a instrução:

```
CALL        0F37DH
```

Uma vez apresentada toda a teoria vamos para a prática com dois exemplos simples. O primeiro cria um arquivo e grava 128 bytes, e o segundo, chama o arquivo gravado.

1º Exemplo: Como gravar um arquivo de 128 bytes

```
1000            ORG     0E000H
1010            DI
1020    FCB:    EQU     0D800H
1030    DMA:    EQU     0D900H
1040    BDOS:   EQU     0F37DH
1050            LD      HL,NOME
1060            LD      DE,FCB
1070            LD      BC,37
1080            LDIR
1090            LD      DE,DMA
```

```
1100          LD      C,1AH
1110          CALL    BDOS
1120          LD      DE,FCB
1130          CALL    EXISTE
1140          LD      HL,MNERR1
1150          JR      Z,ERRO
1160          LD      C,16H
1170          PUSH    DE
1180          CALL    BDOS
1190          POP     DE
1200          INC     A
1210          LD      HL,MNERR2
1220          JR      Z,ERRO
1230          LD      C,15H
1240          PUSH    DE
1250          CALL    BDOS
1260          POP     DE
1270          OR      A
1280          LD      HL,MNERR3
1290          JR      NZ,ERRO
1300          LD      C,10H
1310          PUSH    DE
1320          CALL    BDOS
1330          POP     DE
1340          OR      A
1350          LD      HL,MNERR3
1360          JR      NZ,ERRO
1370          EI
1380          RET
1390   ERRO:  LD      B,00H
1400          PUSH    HL
1410   LOOP:  LD      A,(HL)
1420          CP      '/'
1430          JR      Z,IMPRIM
1440          INC     HL
```

```
1450           INC      B
1460           JR       LOOP
1470  IMPRIM:  POP      HL
1480           LD       DE,(0F7F8H)
1490           EX       DE,HL
1500           LD       (HL),B
1510           INC      HL
1520           LD       (HL),E
1530           INC      HL
1540           LD       (HL),D
1550           EI
1560           RET
1570  EXISTE:  PUSH     DE
1580           LD       C,0FH
1590           CALL     BDOS
1600           OR       A
1610           POP      DE
1620           RET
1630  NOME:    DEFB     01H,'DISKETTETST'
1640           DEFW     0,0,0,0,0,0,0,0,0,0,0,0
1650  MNERR1:  DEFM     'Este arquivo já existe. /'
1660  MNERR2:  DEFM     'O diretório está cheio. /'
1670  MNERR3:  DEFM     'Não há espaço no disco. /'
```

O programa acima cria um arquivo e depois grava nele um bloco de 128 bytes, vamos ver como isso acontece. Da linha 1020 a linha 1110 informamos ao sistema o endereço do FCB e do DMA, como também o nome do arquivo a ser criado. Da linha 1120 a 1150 verificamos se por acaso o arquivo a ser criado já existe, evitando assim a perda acidental de um programa. Da linha 1160 a 1220 criamos o arquivo e verificamos se o diretório já não está cheio. Da linha 1230 a 1290 gravamos os 128 bytes presentes a partir do DMA e verificamos se não houve falta de espaço no disco. Da linha 1300 a 1350 fechamos o arquivo, onde por conveniência aproveitamos a mensagem de erro da situação anterior. Nas linha 1370 e 1380 preparamos a volta ao BASIC. Note que as mensagens de erro são enviadas como parâmetro da fun-

ção USR, logo para testar o programa acima compile-o e depois faça rodar o programa em BASIC a seguir:

```
10  DEFUSR=&HE000
20  A$=USR("MSX")
30  PRINT A$
```

Se você quiser incrementar ainda mais este programa passe o nome do arquivo a ser criado como argumento da função USR. Note ainda que o nome do arquivo no programa assembly está escrito como "DISKETTETST" e não como "DISKETTE.TST". Outro detalhe é que você deve preencher os 37 bytes que compõem o FCB com zeros antes de colocar o nome do arquivo.

2º. Exemplo: Como carregar um bloco de 120 bytes

```
1000            ORG     0E000H
1010            DI
1020    FCB:    EQU     0D800H
1030    DMA:    EQU     0D900H
1040    BDOS:   EQU     0F37DH
1050            LD      HL,NOME
1060            LD      DE,FCB
1070            LD      BC,37
1080            LDIR
1090            LD      DE,DMA
1100            LD      C,1AH
1110            CALL    BDOS
1120            LD      DE,FCB
1130            LD      C,0FH
1140            PUSH    DE
1150            CALL    EXISTE
1160            POP     DE
1170            LD      HL,MNERR1
1180            JR      NZ,ERRO
```

```
1190          LD       C,14H
1200          PUSH     DE
1210          CALL     BDOS
1220          POP      DE
1230          CP       02H
1240          LD       HL,MNERR2
1250          JR       Z,ERRO
1260          LD       C,10H
1270          PUSH     DE
1280          CALL     BDOS
1290          POP      DE
1300          OR       A
1310          LD       HL,MNERR2
1320          JR       NZ,ERRO
1330          EI
1340          RET
1350 ERRO:    LD       B,00H
1360          PUSH     HL
1370 LOOP:    LD       A,(HL)
1380          CP       '/'
1390          JR       Z,IMPRIM
1400          INC      HL
1410          INC      B
1420          JR       LOOP
1430 IMPRIM:  POP      HL
1440          LD       DE,(0F7F8H)
1450          EX       DE,HL
1460          LD       (HL),B
1470          INC      HL
1480          LD       (HL),E
1490          INC      HL
1500          LD       (HL),D
1510          EI
1520          RET
1530 EXISTE:  LD       C,0FH
```

```
1540            CALL   BDOS
1550            OR     A
1560            RET
1570  NOME:     DEFB   01H,'DISKETTETST'
1580            DEFW   0,0,0,0,0,0,0,0,0,0,0,0
1590  MNERR1:   DEFM   'Este arquivo não existe. /'
1600  MNERR2:   DEFM   'Erro de leitura. /'
```

O funcionamento deste programa é semelhante, e valem as observações feitas no final dos comentários sobre o programa anterior. Da linha 1020 a 1110 informamos ao sistema a posição inicial do FCB, do DMA e o nome do arquivo a ser carregado. Da linha 1120 a 1180 verificamos se o arquivo existe, se existir preparamos o arquivo para leitura. Da linha 1190 a 1250 lemos um bloco de 128 bytes e verificamos se não houve qualquer problema no carregamento. O bloco carregado está agora armazenado em RAM a partir do endereço do DMA. Da linha 1260 a 1320 fechamos o arquivo, e finalmente preparamos a volta ao BASIC nas linha 1330 e 1340. Note que a mensagem de erro para fechamento de arquivo foi mantida igual a de "problema no carregamento", apenas por conveniência. Se você quiser ser mais correto coloque uma mensagem do tipo "Problema no fechamento do arquivo.". Para testar o programa acima compile-o e faça rodar o seguinte programa em BASIC:

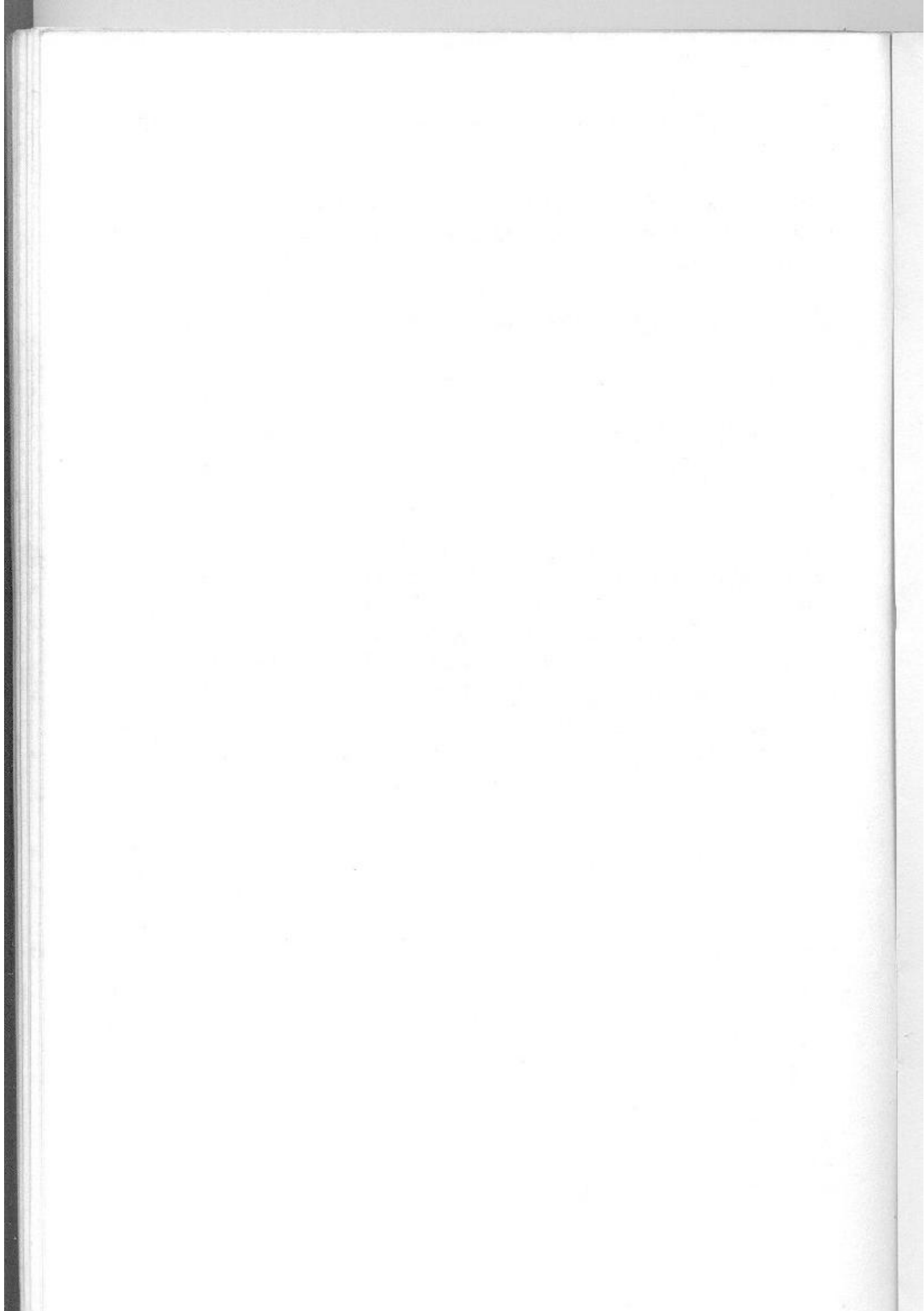
```
10  DEFUSR=&HE000
20  A$=USR("MSX")
30  PRINT A$
```

Com este capítulo fechamos a nossa introdução ao DISK DRIVE usando o DISK BASIC. Como você deve ter notado este é um assunto amplo demais para ser tratado apenas num capítulo, de modo que, aguarde informações mais detalhadas no próximo livro sobre programação avançada. Antes de encerrar este capítulo gostaria de fazer dois comentários sobre os exemplos aqui expostos. O primeiro se refe-

re ao programa BASIC utilizado para ler o diretório. Da forma que está este programa irá imprimir também o nome dos arquivos já deletados, se você não desejar que isso aconteça acrescente a linha abaixo:

145 IF PEEK (A)=&HE5 THEN GOTO 290

O outro comentário é para que você não se esqueça de preencher os 37 bytes de FCB com zeros, antes de colocar nele o nome do arquivo a ser criado, desculpe insistir tanto neste assunto mas é que ao fazer os exemplos acima escolhi uma área que normalmente está preenchida com zeros.

Agora terminamos todos os assuntos que este livro se propos a apresentar. Alguns assuntos que não tratamos com devida profundidade estão expostos nos apêndices a seguir. Se você tiver alguma dúvida ou quiser emitir a sua opinião sobre este livro, sinta-se à vontade para fazê-lo escrevendo para a **CIÊNCIA MODERNA COMPUTAÇÃO**, cujo endereço se encontra no ínicio deste livro.

APÊNDICE A

Neste apêndice iremos apresentar com mais detalhes, a passagem de argumentos alfanuméricos do BASIC para rotinas em linguagem de máquina e vice-versa.

No Capítulo 14 mostramos como passar argumentos inteiros do BASIC para as rotinas em linguagem de máquina e vice-versa, além disso, apresentamos os conceitos gerais de como fazer o mesmo com argumentos alfanuméricos. Como você deve estar lembrado, afirmamos que o endereço F663H indicava ao sistema que o argumento da função USR era alfanumérico cuando assumia o valor 3. Da mesma forma os endereços F7F8H e F7F9H continham um endereço de memória que por sua vez apontava para uma zona de trabalho onde o primeiro byte continha o número de caracteres da string do argumento, e os segundo e terceiro bytes, continham o endereço do início da string na memória. No capítulo sobre rotinas de I/O para disk drive, utilizamos estes conceitos para imprimir as mensagens de erro. Vejamos agora um exemplo para você fixar melhor estes conceitos.

```
1000          ORG     0E000H
1010          DI
1020          LD      HL,(0F7F8H)
1030          LD      B,(HL)
1040          INC     HL
1050          LD      E,(HL)
1060          INC     HL
1070          LD      D,(HL)
1080          PUSH    DE
1090          POP     HL
1100          PUSH    BC
1110          LD      C,B
1120          LD      B,00H
1130          ADD     HL,BC
1140          DEC     HL
1150          POP     BC
1160          SRA     B
1170  LOOP:   LD      A,(HL)
1180          PUSH    AF
1190          LD      A,(DE)
1200          LD      (HL),A
1210          POP     AF
1220          LD      (DE),A
1230          INC     DE
1240          DEC     HL
1250          DJNZ    LOOP
1260          EI
1270          RET
```

Vamos então examinar o funcionamento do programa. Na linha 1020 pegamos o endereço da zona de trabalho. Da linha 1030 a linha 1070 pegamos o tamanho da string e colocamos no registro B, e logo depois colocamos no par DE o endereço inicial da string na memória. Da linha 1080 a linha 1140 calculamos o endereço final da string baseado no seu endereço inicial e no seu comprimento, o resultado é então colo-

cado no par HL. Na linha 1160 dividimos o comprimento da string por dois. Da linha 1170 a 1250 processamos a troca de caracteres, trocando o último caracter com o primeiro, o penúltimo com o segundo e assim por diante. Como fazemos duas substituições por vez tivemos de dividir o registro B por dois na linha 1160. Para testar o programa compile-o e depois rode o programa BASIC a seguir:

```
10  DEFUSR=&HE000
20  CLS:KEYOFF
30  LOCATE 0,10:INPUT "Qual é a palavra";A$
40  CLS:LOCATE 0,10:PRINT A$;TAB(20);USR(A$)
50  IF INKEY$=" " THEN 50 ELSE GOTO 10
```

Esta rotina em linguagem de máquina pode ser usada para testar se uma palavra é palindroma ou não.

APÊNDICE B

Este apêndice irá apresentar algumas das intruções secretas. Assim sendo iremos apresentar somente as instruções de carregamento que dividem os pares IX e IY em dois registros de oito bits, designados respectivamente por HX, LX e HY e LY. A tabela no final deste apêndice mostra as novas instruções disponíveis.

Para usá-las nos montadores comuns, como no SIMPLE ASM V2.1, utilizado por nós, faça:

```
LD HX,9:DEFB    0DDH
        LD      H,9

INC LX:DEFB     0DDH
       INC      L

LD HY,8:DEFB    0FDH
        LD      H,8
```

Deste modo você poderá dispor de mais 46 instruções que lhe poderão ser bastante úteis, principalmente na proteção de software.

CÓDIGO (HEX)	INSTRUÇÃO	CÓDIGO (HEX)	INSTRUÇÃO
DD24	INC HX	DD6A	LD LX,D
DD25	DEC HX	DD6B	LD LX,E
DD26nn	LD HX,nn	DD6C	LD LX,HX
DD2C	INC LX	DD6D	LD LX,LX
DD2D	DEC LX	DD6F	LD LX,A
DD2Enn	LD LX,nn	DD7C	LD A,HX
DD44	LD B,HX	DD7D	LD A,LX
DD45	LD B,LX	DD84	ADD A,HX
DD4C	LD C,HX	DD85	ADD A,LX
DD4D	LD C,LX	DD8C	ADC A,HX
DD54	LD D,HX	DD8D	ADC A,LX
DD55	LD D,LX	DD94	SUB HX
DD5C	LD E,HX	DD95	SUB LX
DD5D	LD E,LX	DD9C	SBC A,HX
DD60	LD HX,B	DD9D	SBC A,LX
DD61	LD HX,C	DDA4	AND HX
DD62	LD HX,D	DDA5	AND LX
DD63	LD HX,E	DDAC	XOR HX
DD64	LD HX,HX	DDAD	XOR LX
DD65	LD HX,LX	DDB4	OR HX
DD67	LD .HX,A	DDB5	OR LX
DD68	LD LX,B	DDBC	CP HX
DD69	LD LX,C	DDBD	CP LX

APÊNDICE C

Neste apêndice iremos mostrar através de exemplos como fazer multiplicações e divisões simples usando a linguagem de máquina.

1º. Exemplo: Como multiplicar números de 8 bits

```
1000             ORG      0E000H
1010             DI
1020             LD       HL,0F7F8H
1030             LD       E,(HL)
1040             LD       D,00H
1050             INC      HL
1060             LD       A,(HL)
1070             LD       HL,0000H
1080             LD       B,08H
1090   MULTIP:   ADD      HL,HL
1100             RLA
1110             JR       NC,LOOP
```

```
1120          ADD     HL,DE
1130  LOOP:   DJNZ    MULTIP
1140          EX      DE,HL
1150          LD      HL,0F7F8H
1160          LD      (HL),E
1170          INC     HL
1180          LD      (HL),D
1190          EI
1200          RET
```

Vamos então entender o funcionamento do programa. Da linha 1020 a linha 1060 pegamos o multiplicando e colocamos no par DE, logo após pegamos o multiplicador e colocamos no registro A. O multiplicando é o valor LSB do argumento da função USR e o multiplicador é o valor MSB da função USR. Na linha 1070 preparamos HL para receber o produto. Na linha 1080 preparamos o registro B como contador de uma multiplicação de dois números de oito bits. Da linha 1090 a linha 1130 fazemos a multiplicação propriamente dita, segundo o algoritmo que afirma que se o presente bit do multiplicador é 1, devemos somar o multiplicando ao produto parcial, que por sua vez está contido em HL. Para que tudo corra bem devemos alinhar os produtos parciais da mesma forma que quando realizamos as operações de multiplicação manualmente. Isto é conseguido examinando o carry do registro A, que nada mais é que um exame do presente bit através da instrução RLA. Se o carry for 1 somamos o multiplicando ao produto na linha 1120 e alinhamos este novo valor na linha 1090 para que as próximas somas fiquem alinhadas também. Ao final da operação de multiplicação tranferimos o resultado para o par DE, colocamos este resultado como argumento da função USR e preparamos a volta ao BASIC. Para testar este programa compile-o e faça rodar o programa BASIC a seguir:

```
10 DEFUSR=&HE000
20 REM ENTRA COM O MULTIPLICADOR
30 A$="08"
40 REM ENTRA COM O MULTIPLICANDO
```

```
50  B$="20"
60  REM PREPARA TRANSFERÊNCIA DO ARGUMENTO
70  A%=VAL("&H"+A$+B$)
80  A%=USR(A%)
90  CLS:LOCATE 0,10:PRINT "PRODUTO =";A%
```

2º. Exemplo: Como dividir um número de 16 bits por outro de 8 bits, ambos sem sinal:

```
1000            ORG     0E000H
1010            DI
1020            LD      HL,0F7F8H
1030            LD      E,(HL)
1040            INC     HL
1050            LD      D,(HL)
1060            LD      A,08H
1070            EX      DE,HL
1080            LD      C,A
1090            LD      B,08H
1100  DIVISA:   ADD     HL,HL
1110            LD      A,H
1120            SUB     C
1130            JR      C,LOOP
1140            LD      H,A
1150            INC     L
1160  LOOP:     DJNZ    DIVISA
1170            EX      DE,HL
1180            LD      HL,0F7F8H
1190            LD      (HL),E
1200            INC     HL
1210            LD      (HL),D
1220            EI
1230            RET
```

A lógica envolvida neste programa é semelhante a envolvida no pro-

grama de multiplicação. O algoritmo empregado testa se o divisor pode ser subtraído dos oito bits mais significativos do dividendo sem vai um (Carry igual a 1), o atual bit do quociente será 1, caso contrário será zero. Do mesmo modo que na multiplicação devemos alinhar o dividendo e o quociente, isto é feito na linha 1100. Da linha 1020 a 1050 transferimos o dividendo para o par DE. Na linha 1060 estabelecemos o divisor no registro A, logo após colocamos o dividendo no par HL, o divisor no registro C e preparamos o registro B para funcionar como um contador. Da linha 1100 a linha 1160 fazemos a divisão. Na linha 1100 alinhamos o dividendo e o quociente, nas linhas 1110, 1120 e 1130 testamos se o divisor pode ser subtraído do dividendo ou não, se puder, subtraímos na linha 1140 e somamos 1 ao quociente na linha 1150. Da linha 1170 a 1230 preparamos a transferência do resultado para a zona de argumentos, e voltamos ao BASIC. Para testar este programa compile-o e rode o programa a seguir:

```
10  DEFUSR=&HE000
20  A%=&H480:A%=USR(A%)
30  A$=RIGHT$("0000"+HEX$(A%),4)
40  CLS:LOCATE 0,10:PRINT "QUOCIENTE=";
50  PRINT RIGHT$(A$,2)"H";TAB(20);
60  PRINT "RESTO =";LEFT$(A$,2)"H"
```

Se você quiser alterar o dividendo basta mudar o valor de A% na linha 20.

APÊNDICE D

Listagem Alfabética

Hexa	Mnemônico		Hexa	Mnemônico	
8E	ADC	A,(HL)	ED7A	ADC	HL,SP
DD8Edo	ADC	A,(IX+d)	86	ADD	A,(HL)
FD8Edo	ADC	A,(IY+d)	DD86do	ADD	A,(IX+d)
8F	ADC	A,A	FD86do	ADD	A,(IY+d)
88	ADC	A,B	87	ADD	A,A
89	ADC	A,C	80	ADD	A,B
8A	ADC	A,D	81	ADD	A,C
8B	ADC	A,E	82	ADD	A,D
8C	ADC	A,H	83	ADD	A,E
8D	ADC	A,L	84	ADD	A,H
CEno	ADC	A,n	85	ADD	A,L
ED4A	ADC	HL,BC	C6no	ADD	A,n
ED5A	ADC	HL,DE	09	ADD	HL,BC
ED6A	ADC	HL,HL	19	ADD	HL,DE

Hexa	Mnemônico	
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA6do	AND	(IX+d)
FDA6do	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L
E6no	AND	n
CB46	BIT	0,(HL)
DDCBdo46	BIT	0,(IX+d)
FDCBdo46	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCBdo4E	BIT	1,(IX+d)
FDCBdo4E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCBdo56	BIT	2,(IX+d)
FDCBdo56	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCBdo5E	BIT	3,(IX+d)
FDCBdo5E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCBdo66	BIT	4,(IX+d)
FDCBdo66	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCBdo6E	BIT	5,(IX+d)
FDCBdo6E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D

Hexa	Mnemônico		Hexa	Mnemônico	
CB6B	BIT	5,E	BA	CP	D
CB6C	BIT	5,H	BB	CP	E
CB6D	BIT	5,L	BC	CP	H
CB76	BIT	6,(HL)	BD	CP	L
DDCBdo76	BIT	6,(IX+d)	FEno	CP	n
FDCBdo76	BIT	6,(IY+d)	EDA9	CPD	
CB77	BIT	6,A	EDB9	CPDR	
CB70	BIT	6,B	EdA1	CPI	
CB71	BIT	6,C	EDB1	CPIR	
CB72	BIT	6,D	2F	CPL	
CB73	BIT	6,E	27	DAA	
CB74	BIT	6,H	35	DEC	(HL)
CB75	BIT	6,L	DD35do	DEC	(IX+d)
CB7E	BIT	7,(HL)	FD35do	DEC	(IY+d)
DDCBdo7E	BIT	7,(IX+d)	3D	DEC	A
FDCBdo7E	BIT	7,(IY+d)	05	DEC	B
CB7F	BIT	7,A	0B	DEC	BC
CB78	BIT	7,B	0D	DEC	C
CB79	BIT	7,C	15	DEC	D
CB7A	BIT	7,D	1B	DEC	DE
CB7B	BIT	7,E	1D	DEC	E
CB7C	BIT	7,H	25	DEC	H
CB7D	BIT	7,L	2B	DEC	HL
DCmono	CALL	C,nm	DD2B	DEC	IX
FCmono	CALL	M,nm	FD2B	DEC	IY
D4mono	CALL	NC,nm	2D	DEC	L
CDmono	CALL	nm	3B	DEC	SP
C4mono	CALL	NZ,nm	F3	DI	
F4mono	CALL	P,nm	10do	DJNZ	d
ECmono	CALL	PE,nm	FB	EI	
E4mono	CALL	PO,nm	E3	EX	(SP),HL
CCmono	CALL	Z,nm	DDE3	EX	(SP),IX
3F	CCF		FDE3	EX	(SP),IY
BE	CP	(HL)	08	EX	AF,AF'
DDBEdo	CP	(IX+d)	EB	EX	DE,HL
FDBEdo	CP	(IY+d)	D9	EXX	
BF	CP	A	76	HALT	
B8	CP	B	ED46	IM	0
B9	CP	C	ED56	IM	1

Hexa	Mnemônico	
ED5E	IM	2
ED78	IN	A,(C)
DBno	IN	A,(n)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD34do	INC	(IX+d)
FD34do	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DAmono	JP	C,nm
FAmono	JP	M,nm
D2mono	JP	NC,nm
C3mono	JP	nm
C2mono	JP	Nz,nm
F2mono	JP	P,nm
EAmono	JP	PE,nm

Hexa	Mnemônico	
E2mono	JP	PO,nm
CAmono	JP	Z,nm
38do	JR	C,d
18do	JR	d
30do	Jr	NC,d
20do	JR	NZ,d
28do	JR	Z,d
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
36no	LD	(HL),n
DD77do	LD	(IX+d),A
DD70do	LD	(IX+d),B
DD71do	LD	(IX+d),C
DD72do	LD	(IX+d),D
DD73do	LD	(IX+d),E
DD74do	LD	(IX+d),H
DD75do	LD	(IX+d),L
DD36dono	LD	(IX+d),n
FD77do	LD	(IY+d),A
FD70do	LD	(IY+d),B
FD71do	LD	(IY+d),C
FD72do	LD	(IY+d),D
FD73do	LD	(IY+d),E
FD74do	LD	(IY+d),H
FD75do	LD	(IY+d),L
FD36dono	LD	(IY+d),n
32mono	LD	(nm),A
ED43mono	LD	(nm),BC
ED53mono	LD	(mn),DE
22mono	LD	(nm),HL
DD22mono	LD	(nm),IX
FD22mono	LD	(nm),IY

Hexa	Mnemônico	
ED73mono	LD	(nm),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)
DD7Edo	LD	A,(IX+d)
FD7Edo	LD	A,(IY+d)
3Amono	LD	A,(nm)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3Eno	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD46do	LD	B,(IX+d)
FD46do	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
06no	LD	B,n
ED4Bmono	LD	BC,(nm)
01mono	LD	BC,mn
4E	LD	C,(HL)
DD4Edo	LD	C,(IX+d)
FD4Edo	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H

Hexa	Mnemônico	
4D	LD	C,L
0Eno	LD	C,n
56	LD	D,(HL)
DD56do	LD	D,(IX+d)
FD56do	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
16no	LD	D,n
ED5Bmono	LD	DE,(nm)
11mono	LD	DE,nm
5E	LD	E,(HL)
DD5Edo	LD	E,(IX+d)
FD5Edo	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1Eno	LD	E,n
66	LD	H,(HL)
DD66do	LD	H,(IX+d)
FD66do	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
26no	LD	H, n
2Amono	LD	HL,(nm)
21mono	LD	HL,nm

Hexa	Mnemônico	
ED47	LD	I,A
DD2Amono	LD	IX,(nm)
DD21mono	LD	IX,nm
FD2Amono	LD	IY,(nm)
FD21mono	LD	IY,nm
6E	LD	L,(HL)
DD6Edo	LD	L,(IX+d)
FD6Edo	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2Eno	LD	L,n
ED4F	LD	R,A
ED7Bmono	LD	SP,(nm)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
31mono	LD	SP,nm
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB6do	OR	(IX+d)
FDB6do	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F6no	OR	n

Hexa	Mnemônico	
EDBB	OTDR	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D3no	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCBdo86	RES	0,(IX+d)
FDCBdo86	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCBdo8E	RES	1,(IX+d)
FDCBdo8E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B

Hexa	Mnemônico	
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCBdo96	RES	2,(IX+d)
FDCBdo96	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB85	RES	2,L
CB9E	RES	3,(HL)
DDCBdo9E	RES	3,(IX+d)
FDCdo9E	RES	3,(IY+d)
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCBdoA6	RES	4,(IX+d)
FDCBdoA	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCBdoAE	RES	5,(IX+d)
FDCBdoAE	RES	5,(IY+d)
CBAF	RES	5,A

Hexa	Mnemônico	
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCBdoB6	RES	6,(IX+d)
FDCBdoB6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCBdoBE	RES	7,(IX+d)
FDCBdoBE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	PO
C8	RET	Z
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCBdo16	RL	(IX+d)

Hexa	Mnemônico	
FDCBdo16	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCBdo06	RLC	(IX+d)
FDCBdo06	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCBdo1E	RR	(IX+d)
FDCBdo1E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCBdo0E	RRC	(IX+d)
FDCBdo0E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D

Hexa	Mnemônico	
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
0F	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
9E	SBC	A,(HL)
DD9Edo	SBC	A,(IX+d)
FD9Edo	SBC	(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
DEno	SBC	A,n
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCBdoC6	SET	0,(IX+d)
FDCBdoC6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L

Hexa	Mnemônico	
CBCE	SET	1,(HL)
DDCBdoCE	SET	1,(IX+d)
FDCBdoCE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCBdoD6	SET	2,(IX+d)
FDCBdoD6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBDE	SET	3,(HL)
DDCBdoDE	SET	3,(IX+d)
FDCBdoDE	SET	3,(IY+d)
CBDF	SET	3,A
CBD8	SET	3,B
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)
DDCBdoE6	SET	4,(IX+d)
FDCBdoE6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H

Hexa	Mnemônico	
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCBdoEE	SET	5,(IX+d)
FDCBdoEE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCBdoF6	SET	6,(IX+d)
FDCBdoF6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCBdoFE	SET	7,(IX+d)
FDCBdoFE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCBdo26	SLA	(IX+d)
FDCBdo26	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E

Hexa	Mnemônico	
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCBdo2E	SRA	(IX+d)
FDCBdo2E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCBdo3E	SRL	(IX+d)
FDCBdo3E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L

Hexa	Mnemônico	
96	SUB	(HL)
DD96do	SUB	(IX+d)
FD96do	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D6no	SUB	n
AE	XOR	(HL)
DDAEdo	XOR	(IX+d)
FDAdo	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EEno	XOR	n

Listagem Numérica

Hexa	Mnemônico	
00	NOP	
01mono	LD	BCnm
02	LD	(BC),A
03	INC	BC
04	INC	B
05	DEC	B
06no	LD	B,n
07	RLCA	
08	EX	AF,AF'

Hexa	Mnemônico	
09	ADD	HL,BC
0A	LD	A,(BC)
0B	DEC	BC
0C	INC	C
0D	DEC	C
0Eno	LD	C, n
0F	RRCA	
10do	DJNZ	d
11mono	LD	DE,nm

Hexa	Mnemônico		Hexa	Mnemônico	
12	LD	(DE),A	39	ADD	HL,SP
13	INC	DE	3Amono	LD	A,(nm)
14	INC	D	3B	DEC	SP
15	DEC	D	3C	INC	A
16no	LD	D,n	3D	DEC	A
17	RLA		3Eno	LD	A,n
18do	JR	d	3F	CCF	
19	ADD	HL,DE	40	LD	B,B
1A	LD	A,(DE)	41	LD	B,C
1B	DEC	DE	42	LD	B,D
1C	INC	E	43	LD	B,E
1D	DEC	E	44	LD	B,H
1Eno	LD	E, n	45	LD	B,L
1F	RRA		46	LD	B,(HL)
20do	JR	NZ,d	47	LD	B,A
21mono	LD	HL,nm	48	LD	C,B
22mono	LD	(nm),HL	49	LD	C,C
23	INC	HL	4A	LD	C,D
24	INC	H	4B	LD	C,E
25	DEC	H	4C	LD	C,H
26no	LD	H,n	4D	LD	C,L
27	DAA		4E	LD	C,(HL)
28do	JR	Z,d	4F	LD	C,A
29	ADD	HL,HL	50	LD	D,B
2Amono	LD	HL,(nm)	51	LD	D,C
2B	DEC	HL	52	LD	D,D
2C	INC	L	53	LD	D,E
2D	DEC	L	54	LD	D,H
2Eno	LD	L,n	55	LD	D,L
2F	CPL		56	LD	D,(HL)
30do	JR	NC,d	57	LD	D,A
31mono	LD	SP,nm	58	LD	E,B
32mono	LD	(nm),A	59	LD	E,C
33	INC	SP	5A	LD	E,D
34	INC	(HL)	5B	LD	E,E
35	DEC	(HL)	5C	LD	E,H
36no	LD	(HL),n	5D	LD	E,L
37	SCF		5E	LD	E,(HL)
38do	JR	C,d	5F	LD	E,A

Hexa	Mnemônico	
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
66	LD	H,(HL)
67	LD	H,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
6E	LD	L,(HL)
6F	LD	L,A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
76	HALT	
77	LD	(HL),A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
7D	LD	A,L
7E	LD	A,(HL)
7F	LD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
86	ADD	A,(HL)

Hexa	Mnemônico	
87	ADD	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
8E	ADC	A,(HL)
8F	ADC	A,A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
96	SUB	(HL)
97	SUB	A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
9E	SBC	A,(HL)
9F	SBC	A,A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L
A6	AND	(HL)
A7	AND	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L

Hexa	Mnemônico	
AE	XOR	(HL)
AF	XOR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
B6	OR	(HL)
B7	OR	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
BE	CP	(HL)
BF	CP	A
C0	RET	NZ
C1	POP	BC
C2mono	JP	NZ,nm
C3mono	JP	nm
C4mono	CALL	NZ,nm
C5	PUSH	BC
C6no	ADD	A,n
C7	RST	00H
C8	RET	Z
C9	RET	
CAmono	JP	Z,nm
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
CB06	RLC	(HL)
CB07	RLC	A
CB08	RRC	B
CB09	RRC	C

Hexa	Mnemônico	
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
CB0E	RRC	(HL)
CB0F	RRC	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
CB16	RL	(HL)
CB17	RL	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
CB1E	RR	(HL)
CB1F	RR	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB26	SLA	(HL)
CB27	SLA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB2E	SRA	(HL)
CB2F	SRA	A
CB38	SRL	B

Hexa	Mnemônico		Hexa	Mnemônico	
CB39	SRL	C	CB60	BIT	4,B
CB3A	SRL	D	CB61	BIT	4,C
CB3B	SRL	E	CB62	BIT	4,D
CB3C	SRL	H	CB63	BIT	4,E
CB3D	SRL	L	CB64	BIT	4,H
CB3E	SRL	(HL)	CB65	BIT	4,L
CB3F	SRL	A	CB66	BIT	4,(HL)
CB40	BIT	0,B	CB67	BIT	4,A
CB41	BIT	0,C	CB68	BIT	5,B
CB42	BIT	0,D	CB69	BIT	5,C
CB43	BIT	0,E	CB6A	BIT	5,D
CB44	BIT	0,H	CB6B	BIT	5,E
CB45	BIT	0,L	CB6C	BIT	5,H
CB46	BIT	0,(HL)	CB6D	BIT	5,L
CB47	BIT	0,A	CB6E	BIT	5,(HL)
CB48	BIT	1,B	CB6F	BIT	5,A
CB49	BIT	1,C	CB70	BIT	6,B
CB4A	BIT	1,D	CB71	BIT	6,C
CB4B	BIT	1,E	CB72	BIT	6,D
CB4C	BIT	1,H	CB73	BIT	6,E
CB4D	BIT	1,L	CB74	BIT	6,H
CB4E	BIT	1,(HL)	CB75	BIT	6,L
CB4F	BIT	1,A	CB76	BIT	6,(HL)
CB50	BIT	2,B	CB77	BIT	6,A
CB51	BIT	2,C	CB78	BIT	7,B
CB52	BIT	2,D	CB79	BIT	7,C
CB53	BIT	2,E	CB7A	BIT	7,D
CB54	BIT	2,H	CB7B	BIT	7,E
CB55	BIT	2,L	CB7C	BIT	7,H
CB56	BIT	2,(HL)	CB7D	BIT	7,L
CB57	BIT	2,A	CB7E	BIT	7,(HL)
CB58	BIT	3,B	CB7F	BIT	7,A
CB59	BIT	3,C	CB80	RES	0,B
CB5A	BIT	3,D	CB81	RES	0,C
CB5B	BIT	3,E	CB82	RES	0,D
CB5C	BIT	3,H	CB83	RES	0,E
CB5D	BIT	3,L	CB84	RES	0,H
CB5E	BIT	3,(HL)	CB85	RES	0,L
CB5F	BIT	3,A	CB86	RES	0,(HL)

Hexa	Mnemônico	
CB87	RES	0,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB8E	RES	1,(HL)
CB8F	RES	1,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB96	RES	2,(HL)
CB97	RES	2,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CB9E	RES	3,(HL)
CB9F	RES	3,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4E
CBA4	RES	4,H
CBA5	RES	4,L
CBA6	RES	4,(HL)
CBA7	RES	4,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L

Hexa	Mnemônico	
CBAE	RES	5,(HL)
CBAF	RES	5,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBB6	RES	6,(HL)
CBB7	RES	6,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
CBBE	RES	7,(HL)
CBBF	RES	7,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBC6	SET	0,(HL)
CBC7	SET	0,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBF4	SET	6,H
CBF5	SET	6,L
CBF6	SET	6,(HL)
CBF7	SET	6,A
CBF8	SET	7,B
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBCE	SET	1,(HL)
CBCF	SET	1,A

Hexa	Mnemônico	
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,F
CBD4	SET	2,H
CBD5	SET	2,L
CBD6	SET	2,(HL)
CBD7	SET	2,A
CBD8	SET	3,B
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBDE	SET	3,(HL)
CBDF	SET	3,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBE6	SET	4,(HL)
CBE7	SET	4,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBEE	SET	5,(HL)
CBEF	SET	5,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,F
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E

Hexa	Mnemônico	
CBFC	SET	7,H
CBFD	SET	7,L
CBFE	SET	7,(HL)
CBFF	SET	7,A
CCmono	CALL	Z,nm
CDmono	CALL	nm
CEno	ADC	A,n
CF	REST	08H
D0	RET	NC
D1	POP	DE
D2mono	JP	NC,nm
D3no	OUT	(n),A
D4mono	CALL	NC,nm
D5	PUSH	DE
D6no	SUB	n
D7	RST	10H
D8	RET	C
D9	EXX	
DAmono	JP	C,nm
DBno	IN	A,(n)
DCmono	CALL	C,nm
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD21mono	LD	IX,nm
DD22monc	LD	(nm),IX
DD23	INC	IX
DD29	ADD	IX,IX
DD2Amono	LD	IX,(nm)
DD2B	DEC	IX
DD34do	INC	(IX+d)
DD35do	DEC	(IX+d)
DD36dono	LD	(IX+d),n
DD39	ADD	IX,SP
DD46do	LD	B,(IX+d)
DD4Edo	LD	C,(IX+d)
DD56do	LD	D,(IX+d)
DD5Edo	LD	E,(IX+d)
DD66do	LD	H,(IX+d)
DD6Edo	LD	L,(IX+d)

Hexa	Mnemônico	
DD70do	LD	(IX+d),B
DD71do	LD	(IX+d),C
DD72do	LD	(IX+d),D
DD73do	LD	(IX+d),E
DD74do	LD	(IX+d),H
DD75do	LD	(IX+d),L
DD77do	LD	(IX+d),A
DD7Edo	LD	A,(IX+d)
DD86do	ADD	A,(IX+d)
DD8Edo	ADC	A,(IX+d)
DD96do	SUB	(IX+d)
DD9Edo	SBC	A,(IX+d)
DDA6do	AND	(IX+d)
DDAEdo	XOR	(IX+d)
DDB6do	OR	(IX+d)
DDBEdo	CP	(IX+d)
DDCBdo06	RLC	(IX+d)
DDCBdo0E	RRC	(IX+d)
DDCBdo16	RL	(IX+d)
DDCBdo1E	RR	(IX+d)
DDCBdo26	SLA	(IX+d)
DDCBdo2E	SRA	(IX+d)
DDCBdo3E	SRL	(IX+d)
DDCBdo46	BIT	0,(IX+d)
DDCBdo4E	BIT	0,(IX+d)
DDCBdo4E	BIT	1,(IX+d)
DDCBdo56	BIT	2,(IX+d)
DDCBdo5E	BIT	3,(IX+d)
DDCBdo66	BIT	4,(IX+d)
DDCBdo6E	BIT	5,(IX+d)
DDCBdo76	BIT	6,(IX+d)
DDCBdo7E	BIT	7,(IX+d)
DDCBdo86	RES	0,(IX+d)
DDCBdo8E	RES	1,(IX+d)
DDCBdo96	RES	2,(IX+d)
DDCBdo9E	RES	3,(IX+d)
DDCBdoA6	RES	4,(IX+d)
DDCBdoAE	RES	5,(IX+d)
DDCBdoB6	RES	6,(IX+d)

Hexa	Mnemônico	
DDCBdoBE	RES	7,(IX+d)
DDCBdo	SET	0,(IX+d)
DDCBdoCE	SET	1,(IX+d)
DDCBdoD6	SET	2,(IX+d)
DDCBdo	SET	3,(IX+d)
DDCBdoE6	SET	4,(IX+d)
DDCBdoEE	SET	5,(IX+d)
DDCBdoF6	SET	6,(IX+d)
DDCBdoFE	SET	7,(IX+d)
DDE1	POP	IX
DDE3	EX	(SP),IX
DDE5	PUSH	IX
DDE9	JP	(IX)
DDF9	LD	SP,IX
DEno	SBC	A,n
DF	REST	18H
E0	RET	PO
E1	POP	HL
E2mono	JP	PO,nm
E3	EX	(SP),HL
E4mono	CALL	PO,nm
E5	PUSH	HL
E6no	AND	n
E7	RST	20H
E8	RET	PE
E9	JP	(HL)
EAmono	JP	PE,nm
EB	EX	DE,HL
ECmono	CALL	PE,nm
ED40	IN	B,(C)
ED41	OUT	(C),B
ED42	SBC	HL,BC
ED43mono	LD	(nm),BC
ED44	NEG	
ED45	RETN	
ED46	IM	0
ED47	LD	I,A
ED48	IN	C,(C)
ED49	OUT	(C),C

Hexa	Mnemônico	
ED4A	ADC	HL,BC
ED4Bmono	LD	BC,(nm)
ED4D	RETI	
ED4F	LD	R,A
ED50	IN	D,(C)
ED51	OUT	(C),D
ED52	SBC	HL,DE
ED53mono	LD	(nm),DE
ED56	IM	1
ED57	LD	A,I
ED58	IN	E,(C)
ED59	OUT	(C),E
ED5A	ADC	HL,DE
ED5Bmono	LD	DE,(nm)
ED5E	IM	2
ED5F	LD	A,R
ED60	IN	H,(C)
ED61	OUT	(C),H
ED62	SBC	HL,HL
ED67	RRD	
ED68	IN	L,(C)
ED69	OUT	(C),L
ED6A	ADC	HL,HL
ED6F	RLD	
ED72	SBC	HL,SP
ED73mono	LD	(nm),SP
ED78	IN	A,(C)
ED79	OUT	(C),A
ED7A	ADC	HL,SP
ED7Bmono	LD	SP,(nm)
EDA0	LDI	
EDA1	CPI	
EDA2	INI	
EDA3	OUTI	
EDA8	LDD	
EDA9	CPD	
EDAA	IND	
EDAB	OUTD	
EDB0	LDIR	

Hexa	Mnemônico	
EDB1	CPIR	
EDB2	INIR	
EDB3	OTIR	
EDB8	LDDR	
EDB9	CPDR	
EDBA	INDR	
EDBB	OTDR	
EEno	XOR	n
EF	RST	28H
F0	RET	P
F1	POP	AF
F2mono	JP	P,nm
F3	DI	
F4mono	CALL	P,nm
F5	PUSH	AF
F6no	OR	n
F7	RST	30H
F8	RET	M
F9	LD	SP,HL
FAmono	JP	M,nm
FB	EI	
FCmono	CALL	M,nm
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD21mono	LD	IY,nm
FD22mono	LD	(nm),IY
FD23	INC	IY
FD29	ADD	IY,IY
FD2Amono	LD	IY,(nm)
FD2B	DEC	IY
FD34do	INC	(IY+d)
FD35do	DEC	(IY+d)
FD36dono	LD	(IY+d),n
FD39	ADD	IY,SP
FD46do	LD	B,(IY+d)
FD4Edo	LD	C,(IY+d)
FD56do	LD	D,(IY+d)
FD5Edo	LD	E,(IY+d)
FD66do	LD	H,(IY+d)

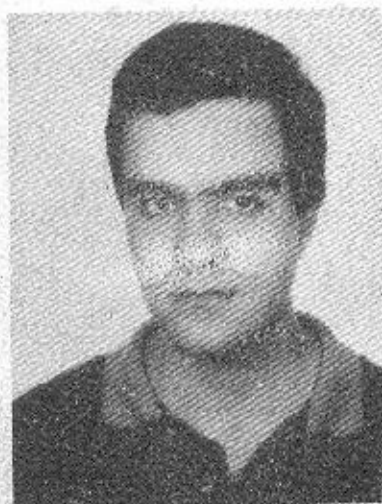
Hexa	Mnemônico	
FD6Edo	LD	L,(IY+d)
FD70do	LD	(IY+d),B
FD71do	LD	(IY+d),C
FD72do	LD	(IY+d),D
FD73do	LD	(IY+d),E
FD74do	LD	(IY+d),H
FD75do	LD	(IY+d),L
FD77do	LD	(IY+d),A
FD7Edo	LD	A,(IY+d)
FD86do	ADD	A,(IY+d)
FD8Edo	ADC	A,(IY+d)
FD96do	SUB	(IY+d)
FD9Edo	SBC	A,(IY+d)
FDA6do	AND	(IY+d)
FDAEdo	XOR	(IY+d)
FDB6do	OR	(IY+d)
FDBEdo	CP	(IY+d)
FDCBdo06	RLC	(IY+d)
FDCBdo0E	RRC	(IY+d)
FDCBdo16	RL	(IY+d)
FDCBdo1E	RR	(IY+d)
FDCBdo26	SLA	(IY+d)
FDCBdo2E	SRA	(IY+d)
FDCBdo3E	SRL	(IY+d)
FDCBdo46	BIT	0,(IY+d)
FDCBdo4E	BIT	1,(IY+d)
FDCBdo56	BIT	2,(IY+d)
FDCBdo5E	BIT	3,(IY+d)
FDCBdo66	BIT	4,(IY+d)
FDCBdo6E	BIT	5,(IY+d)
FDCBdo76	BIT	6,(IY+d)
FDCBdo7E	BIT	7,(IY+d)
FDCBdo86	RES	0,(IY+d)
FDCBdo8E	RES	1,(IY+d)
FDCBdo96	RES	2,(IY+d)
FDCBdo9E	RES	3,(IY+d)
FDCBdoA6	RES	4,(IY+d)
FDCBdoAE	RES	5,(IY+d)
FDCBdoB6	RES	6,(IY+d)
FDCBdoBE	RES	7,(IY+d)
FDCBdoC6	SET	0,(IY+d)
FDCBdoCE	SET	1,(IY+d)
FDCBdoD6	SET	2,(IY+d)
FDCBdoDE	SET	3,(IY+d)
FDCBdoE6	SET	4,(IY+d)
FDCBdoEE	SET	5,(IY+d)
FDCBdoF6	SET	6,(IY+d)
FDCBdoFE	SET	7,(IY+d)
FDE1	POP	IY
FDE3	EX	(SP),IY
FDE5	PUSH	IY
FDE9	JP	(IY)
FDF9	LD	SP,IY
FEno	CP	n
FF	RST	38H

BIBLIOGRAFIA

- Z80 ASSEMBLY LANGUAGE PROGRAMMING
  LANCE A. LEVENTHAL
  OSBORNE/MCGRAW-HILL U.S.A., 1979

- Z80 ASSEMBLY LANGUAGE SUBROTINES
  LANCE A. LEVENTHAL & WINTHROP SAVILLE
  OSBORNE/MCGRAW-HILL U.S.A., 1983

- THE MSX RED BOOK
  AVALON SOFTWARE
  KUMA COMPUTERS LTD. ENGLAND, 1985

- CP/M GUIA DO USUÁRIO
  THOM HOGAN
  OSBORNE/MCGRAW-HILL DO BRASIL LTDA, 1983

Eduardo Alberto R. S. Barbosa é Engenheiro Eletricista formado pela Pontifícia Universidade Católica do Rio de Janeiro.

O interesse pela programação em assembly iniciou-se quando ao estudar a cadeira de Sistemas de Computação percebeu todas as potencialidades oferecidas pelo melhor microprocessador de 8 bits, o Z80. O primeiro passo foi comprar um microcomputador pessoal baseado no Z80. Na época, final de 83, o micro pessoal mais acessível era o TK-82C. Foi neste computador que o autor começou a desvendar, os segredos da linguagem de máquina.

Um ano após, Eduardo Alberto Barbosa, adquiriu um microcomputador DGT-100, com o qual desenvolveu ainda mais a programação em linguagem de máquina. O interesse pela linha MSX, veio no final de 85, com um convite da empresa CIBERNE para que o autor integrasse uma equipe de programadores destinada a desenvolver software nacional para o MSX.

Um ano após, o autor, juntamente com dois amigos, fundou a empresa TURBO EQUIPAMENTOS ELETRÔNICOS LTDA, ficando então responsável pela divisão de software, que atende exclusivamente a linha MSX.